Detecting Unsolvability Based on Parity Functions

Master's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
https://ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Salomé Eriksson

Remo Christen
remo.christen@stud.unibas.ch
2015-051-469

March 13, 2021

Acknowledgments

I am grateful to Prof. Dr. Malte Helmert for the opportunity to write this thesis in the Artificial Intelligence research group. I want to thank Dr. Salomé Eriksson for her guidance in all stages of this thesis. Her explanations, ideas, suggestions, and continuous feedback were invaluable and led to countless realizations and improvements.

I also want to express my gratitude towards my parents, who have always enabled and encouraged me on my path. Last but definitely not least, I want to thank Irene for supporting me in any way possible and giving me the strength and courage to do my best.

Abstract

Unsolvability is an important result in classical planning and has seen increased interest in recent years. This thesis explores unsolvability detection by automatically generating parity arguments, a well-known way of proving unsolvability. The argument requires an invariant measure, whose parity remains constant across all reachable states, while all goal states are of the opposite parity. We express parity arguments using potential functions in the field F_2. We develop a set of constraints that describes potential functions with the necessary separating property, and show that the constraints can be represented efficiently for up to two-dimensional features. Enhanced with mutex information, an algorithm is formed that tests whether a parity function exists for a given planning task. The existence of such a function proves the task unsolvable. To determine its practical use, we empirically evaluate our approach on a benchmark of unsolvable problems and compare its performance to a state of the art unsolvability planner. We lastly analyze the arguments found by our algorithm to confirm their validity, and understand their expressive power.

Table of Contents

1
Introduction

The 15 puzzle was invented over 140 years ago and is still well-known today. In its original form, it consisted of 15 numbered blocks that could be arranged in a 4 by 4 frame, leaving one spot empty. The game is set up by scrambling the blocks and putting them back into the frame randomly. The player must then rearrange the blocks into the ordered goal configuration, as shown in Figure 1.1, by sliding blocks into the empty spot.

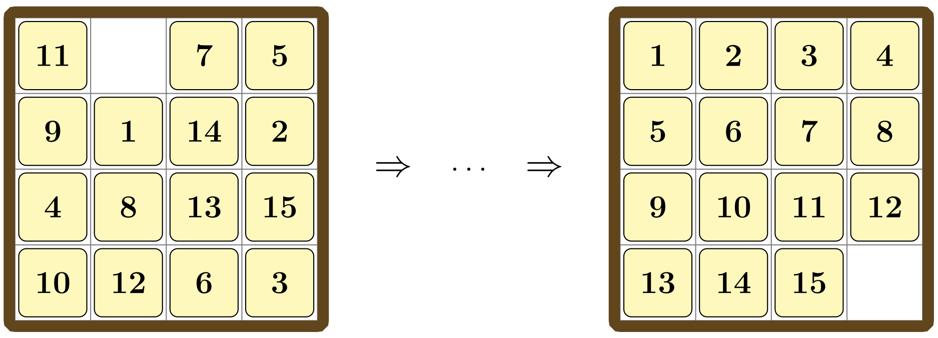


Figure 1.1: Scrambled and traditional goal state.

Problems of this kind are the subject of study in *classical planning*, a branch of model-based artificial intelligence. More generally, planning is the process of finding a course of action which, starting in a defined world state, changes the world in such a way that a set of predefined goals become true. In the classical setting, the world must be deterministic and fully observable.

The 15 puzzle is an example for such a world. A world at the center of the puzzling universe of 1880, when countless people around the globe became captivated by the puzzle's deceptive simplicity. The driving force for this craze was the elusive nature of its solution, which could come naturally one round, only to seem unobtainable the next. Cash prizes have been offered on multiple occasions for a solution to a particularly trivial-looking challenge called the 14-15 puzzle, pictured in Figure 1.2. None of the prizes have ever been claimed, because the challenge is indeed impossible. In fact, the puzzle is unsolvable from half of all scrambled starting positions, as has been shown by Johnson and Story (1879) already before the height of the craze.

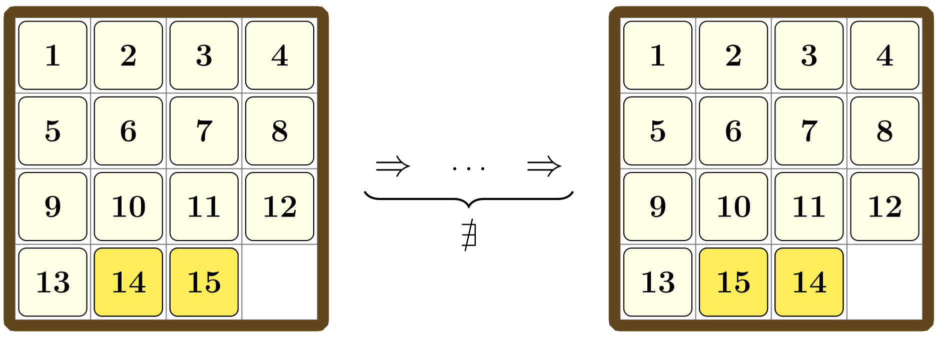


Figure 1.2: Initial and goal state of the unsolvable 14-15 puzzle.

This example shows us that a solution is not the only interesting outcome of planning tasks, but that proving one to be unsolvable is also a meaningful result. Consequently, various approaches to unsolvability have been pursued, with two examples being a specialized merge-and-shrink heuristic by Hoffmann et al. (2014), and using consistency checking concepts on variable subsets by Bäckström et al. (2013).

In this thesis we explore parity arguments as an approach to unsolvability detection. The basic idea of parity arguments is simple: assign unequal parity to starting and goal states and show that this gap cannot be bridged by any action. This conceptually divides a task's state space into two parts: one containing the initial, and the other containing all goal states. The 15 puzzle unsolvability proof by Johnson and Story (1879) is of this form and is probably the most famous application of a parity argument. We take a closer look in the next chapter.

Domain-independence is an essential part of automated planning, and handcrafted proofs for single domains or even tasks are not helpful. Instead, we want to devise a strategy to compute parity arguments for various types of problems automatically. Based on a real-valued approach by Pommerening (2017), we aim to construct a set of constraints that describes parity arguments. A solution to said constraints should return a so called *potential function* that assigns values to states in the way of a parity argument. The existence of such a function would prove a task unsolvable.

To achieve this, we take advantage of the simple even-odd structure of parity by working in F_2, the field with only two elements, and solving constraints using Gaussian elimination instead of a complex LP solver.

This work contains theoretical results about the construction and feasibility of parity constraints, as well as instructions on how to apply these results in practice. We compare three implementations of the resulting algorithm and empirically evaluate its performance on a diverse unsolvability benchmark. This also includes a side by side comparison with Aidos, a state of the art unsolvability planner and winner of the Unsolvability International Planning Competition 2016. Finally, we take a close look at a domain modelling peg solitaire, another classic single player board game, and try to understand and explain the arguments found by our algorithm.

2 Background

In this section we introduce the concepts that are at the core of this thesis. We define our universe, describe the problem we attempt to solve within it, and present the tools required to do so.

2.1 Planning

Automated planning is the discipline of finding a sequence of actions that leads from a given initial state to a goal state. Problem instances for which such a sequence should be found are called planning tasks. They provide a state space as well as initial and goal states. Throughout this thesis we will use the 15 puzzle as an example task to illustrate the presented concepts.

There are multiple ways to formally define planning tasks. In this thesis we will use the SAS$^+$ formalism first proposed by Bäckström and Nebel (1995). We first define the necessary components to then formally construct a planning task.

Definition 1 (Variable)**.** A variable V is defined over a finite domain $d = dom(V)$ and can be assigned a single value $v \in d$.

In upcoming definitions we will refer to some finite set of variables $\mathcal{V}$. Such a set would usually stem from the definition of a *planning task*.

The 15 puzzle can be modeled using 16 variables, one for every cell of the grid. We choose the variable names as the cell's coordinates with the origin in the top left corner. The variables modeling the marked cells in Figure 2.1 would therefore be called, from top to bottom: $\{$🟨 $: V_{0,0},$ 🟧 $: V_{3,1},$ 🟥 $: V_{1,2}\}$. All variables are defined over the same domain d containing the numbers 1 to 15, representing the numbered tiles, as well as a symbol for the blank cell: $d = \{1, 2, \ldots, 15, \Box\}$.

Definition 2 (Atom)**.** An atom A is an assignment $V \mapsto v$ where V is a variable and v a value with $v \in dom(V)$.

We will use the notation $var(A)$ to refer to the variable, and $val(A)$ to refer to the value of atom A. Further, we will denote the set of all atoms in a planning task Π with $\mathcal{A}$. In

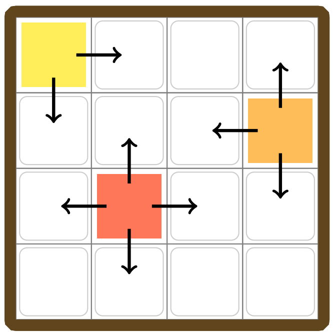


Figure 2.1: Three positions with varying numbers of adjacent cells.

some contexts, we will view variables as sets of atoms, containing an atom for every value in $dom(V)$.

The following are examples for atoms that exist within the 15 puzzle: $\{V_{0,0} \mapsto 5, V_{0,3} \mapsto 11, V_{2,1} \mapsto \square\}$.

Definition 3 (State)**.** A state is a variable assignment of all variables $V \in \mathcal{V}$ to values v and forms a set $\{V \mapsto v \mid V \in \mathcal{V}, v \in dom(V)\}$.

Partial states are variable assignments over a subset of $\mathcal{V}$. We denote the set of variables over which a partial state s is defined by $vars(s) \subseteq \mathcal{V}$. We refer to the value v assigned to a variable V in (partial) state s through $s[V] = v$. We say that $A = \{V \mapsto v\}$ *holds* in s. All possible states of a planning task form its *state space*.

We could model the traditional solved state of the 15 puzzle by the following assignments: $\{V_{0,0} \mapsto 1, V_{1,0} \mapsto 2, \ldots, V_{3,3} \mapsto \square\}$.

Definition 4 (Operator)**.** An operator is a tuple $o = \langle p, e\rangle$ where p are the preconditions and e the effects of o, written as $pre(o)$ and $eff(o)$. Both p and e are partial states over $\mathcal{V}$.

Operator o is *applicable* in state s if the partial state p of o is consistent with s, meaning that the variable assignments given in p also hold in s. Applying operator o in s yields a *successor state* s' of s, written as $s' = s[\![o]\!]$. Variables V in state s' are assigned to $eff(o)[V]$ if $V \in vars(eff(o))$, and to $s[V]$ otherwise. Operators have an associated cost, and when not stated otherwise we assume unit cost: $cost(o) = 1$ for all operators o.

The only kind of operator present in the 15 puzzle is swapping the position of the blank cell with an orthogonally adjacent tile. We thus have to define one such operator for every combination of blank cell and adjacent tile. The number of possible swaps for corner, edge, and center positions of the blank cell are illustrated in Figure 2.1. All operators consist of two preconditions and two effects each. The preconditions ensure that the blank cell and some tile are in adjacent cells. The effects assign the blank cell's variable to the tile it is swapped with and vice versa – the variable representing the tile's cell is assigned to the blank cell. A simple example of an applied operator is shown in Figure 2.2.

Definition 5 (Initial State)**.** An initial state is a state marking the starting point of a planning task.

States that can be reached from the initial state by applying operators are called *reachable* states.

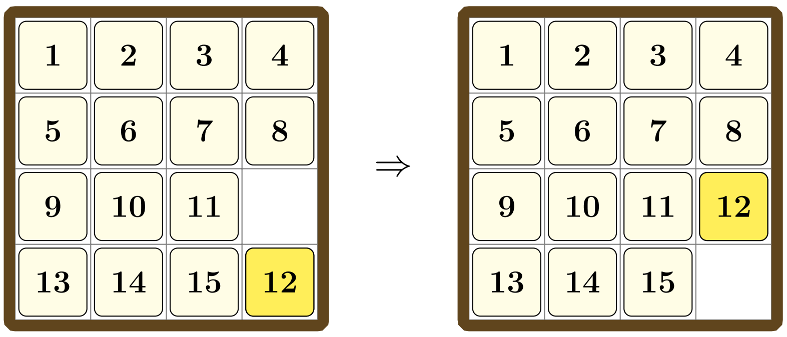


Figure 2.2: Example for applied operator.

For the 15 puzzle, any random permutation of tiles can be chosen as the initial state.

Definition 6 (Goal)**.** A goal is a partial state describing the state or states that must be reached from the initial state.

The traditional goal state for the 15 puzzle is shown in Figure 1.1. We will later see how the choice of initial and goal state determines the solvability of a 15 puzzle instance.

With that we have all components needed to define a planning task.

Definition 7 (Planning Task)**.** A SAS$^+$ planning task is defined by the 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ where $\mathcal{V}$ is a finite set of variables, $\mathcal{O}$ is a finite set of operators, s_0 is an initial state and s_* is a goal.

For most of this thesis we will focus on planning tasks in a particular form.

Definition 8 (Transition Normal Form)**.** For a planning task Π in transition normal form (TNF) it must hold that $vars(pre(o)) = vars(eff(o))$ for all operators o and that the goal is a single, fully defined state.

For operators o in TNF we will use $vars(o) = vars(pre(o)) = vars(eff(o))$. SAS$^+$ tasks can be efficiently transformed to TNF. In short, the idea is to introduce a new value *unknown* to all variables. For every fact a forget-operator is introduced that changes the fact's variable to *unknown* with 0 cost. Conceptually, we can now say that whenever a variable has no value (or we don't care about its value) in the original SAS$^+$ task, it has the value *unknown* in TNF. For a comprehensive account, refer to Pommerening and Helmert (2015) where TNF is first proposed.

Interestingly, the 15 puzzle in general and our example representation in particular already meet the criteria for being in TNF, no transformation needed.

Given a planning task in whichever form, the aim is naturally to find a solution or *plan* for that task. In order to understand plans we first look at a general *operator sequence* $\pi = \langle o_1, \ldots, o_n \rangle$. Applying π to a state s is equivalent to the sequential application of operators in π to s, so that $s[\![\pi]\!] = s[\![o_1]\!] \ldots [\![o_n]\!]$. We call an operator sequence *valid* with respect to a starting state s, if o_1 is applicable in s and all o_i for $2 \leq i \leq n$ are applicable in state $s[\![o_1]\!] \ldots [\![o_{i-1}]\!]$.

Definition 9 (Plan)**.** A plan for planning task Π is an operator sequence π such that the state $s_0[\![\pi]\!]$ is consistent with goal s_*, which requires π to be valid with respect to the initial state s_0.

When talking about the validity of a plan, we always refer to its validity with respect to the initial state.

Planning tasks can be very large, often making unguided search an infeasible strategy for finding plans. *Heuristics* are a way to guide the search – they are functions that estimate the cost of an optimal plan from state s. This measure can then be used to inform the search and guide it towards the goal. While any function that maps states to non-negative numbers (or infinity) is a heuristic, some have properties that make them more useful.

The *perfect heuristic* h^* is the function that returns the true goal distance from a given state s, and returns infinity if the goal is not reachable from s. A heuristic h is *safe* if it only assigns infinity to state s when $h^*(s) = \infty$, *goal-aware* if it assigns 0 to all goal states, *consistent* if $h(s) \leq cost(o) + h(s')$ for all transitions $s \xrightarrow{o} s'$, and *admissible* if $h(s) \leq h^*(s)$ for all states s. These properties are connected in the following way: any goal-aware and consistent heuristic is admissible, and any admissible heuristic is safe and goal-aware.

Even though optimal or sub-optimal plans are what is usually sought when considering planning tasks, there is another informative outcome solvers may produce.

2.2 Unsolvability

Ideally a search has two possible outcomes: either a valid plan or assurance that the task is unsolvable. Much of the research effort in automated planning has been focused on finding plans efficiently more so than to detect unsolvability. Improvements that lead to better performance in finding plans often do not translate to unsolvability detection. While heuristics can make use of dead-end states, the fact remains that they are optimized to find solutions for solvable tasks.

Despite this bias, unsolvability has seen rising interest in recent years. A major milestone in this development is the Unsolvability International Planning Competition (Unsolvability IPC) that was organized, for the first time, in 2016. As a counterpart to the long running International Planning Competition (IPC), it encouraged the development of state of the art planners focused on unsolvability. While this thesis shares the same goal as the Unsolvability IPC competitors, our approach differs from many common strategies. Most notably, our approach does not rely on a search component, but instead aims to exploit inherent characteristics of planning tasks. Invariants are such a characteristic and are essential to the approach of this thesis.

Generally, an invariant can be described as a property over all states of a planning task for which it holds that, if it is true in state s, it is also true in state $s' = s[\![o]\!]$ for all operators o applicable in s. This means that once an invariant is achieved, its truth is preserved across all transitions. In the context of unsolvability, we are primarily interested in properties of the initial state. For this reason we will work with a more specific notion of invariants in this thesis.

Definition 10 (Strong s_0-Invariants)**.** A strong s_0-invariant is a formula I_s, defined over all states of a planning task Π, whose truth value remains constant under operator application. This means that $I_s(s) = I_s(s')$ for all transitions $s \xrightarrow{o} s'$. Additionally, I_s holds true in the initial state.

We say that I_s is *closed* under operator application. We call this first definition strong because it strictly separates the state space into two parts: one where I_s holds and one where it does not. While this kind of invariant is desirable for its expressiveness, it is not always possible to make such strong statements. With the next invariant definition we enable ourselves to make weaker, but still very helpful assertions.

Definition 11 (Weak s_0-Invariants)**.** A weak s_0-invariant is a formula I_w, defined over all states of a planning task Π, whose truth, but not falsity, remains constant under operator application. This means that, if $I_w(s)$ holds, $I_w(s) = I_w(s')$ for all transitions $s \xrightarrow{o} s'$. Additionally, I_w holds true in the initial state.

As opposed to the previous definition, the state space is no longer strictly divided. It is now possible for a transition to lead from a state s where $I_w(s)$ does not hold to one where it does hold. It is important to note that, technically, the truth value of I_w is still constant for all reachable states, because truth is preserved and I_w is guaranteed to be true. We can say that I_w is closed under operator application within the reachable state space. We may at times refer to s_o-invariants without specifying the strong or weak variant. In those cases, the statement applies to both variants.

A simple but powerful type of invariant are mutual exclusion (*mutex*) constraints. They make statements about sets of atoms that can never hold simultaneously. As with invariants, we define mutexes according to the scope needed for this thesis. In particular we limit ourselves to sets of two atoms.

Definition 12 (Mutex)**.** A mutex is a weak s_0-invariant of the form $\neg(a_0 \wedge a_1)$ where a_0 and a_1 are atoms.

Mutexes are a well-known and studied type of invariant. We will consider them in more detail when discussing how we generate mutexes in Section 4.3.

For the 15 puzzle the following formula is a mutex: $\neg(\{V_{0,0} \mapsto 1\} \wedge \{V_{0,1} \mapsto 1\})$. The formula states that cell $(0,0)$ and cell $(0,1)$ can never simultaneously contain tile 1. It is easy to see that this situation cannot occur if we start in a valid state and only perform legal moves.

Invariants have been tied to unsolvability by, for example, Eriksson et al. (2017) who defined *inductive sets*, a generalized notion of invariants, to construct certificates for unsolvability. Another example by Lipovetzky et al. (2016) puts forward an algorithm to compute *traps*, formulas defining inductive sets, and use them to generate invariants and dead-ends.

2.3 Parity Argument for the 14-15 Puzzle

As we have seen in the introduction, parity arguments are a way to prove unsolvability. Let us briefly recap the general structure, applied to a planning task Π. We must show the following two conditions for a given numerical function f defined over all states:

- The parity of $f(s_0)$ is not equal to $f(s_*)$.
- For every transition $s \xrightarrow{o} s'$ it holds that the parity of $f(s)$ is equal to the parity of $f(s')$.

Unsolvability of Π is proven by the existence of a function f satisfying the conditions above. Let us now consider an example for such a function for the 14-15 puzzle.

Johnson and Story (1879) first presented a parity argument that shows the unsolvability of the 14-15 puzzle. A simpler and more modern version of the proof was published by Archer (1999). In the following we will outline Archer's proof in less formal terms.

In order to discuss the unsolvability result for the 14-15 puzzle, we must first introduce some terminology. We refer to the 16 positions on the grid as *cells*. The cells are occupied by one movable *tile* each. We will treat the blank cell as a blank tile here. While we already presented a way to define the state of the 15 puzzle in Section 2.1, we introduce an alternative definition here to get a convenient representation for the proof. In order to keep the two representations separate, we will call the one used in the context of this proof *placement* instead of state.

To define a placement we first determine a total ordering over all cells. This ordering is shown in Figure 2.3. A placement is then defined as the sequence of tiles (numbered and

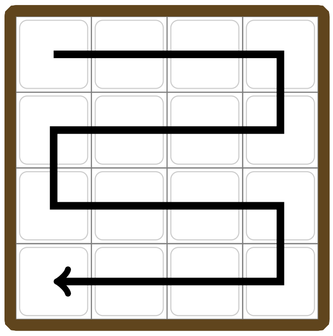


Figure 2.3: Total ordering that defines placements.

blank) according to their position on the path given by the cell ordering. We would write the initial placement of the 14-15 puzzle as:

$$P_{init} = [1, 2, 3, 4, \;\; 8, 7, 6, 5, \;\; 9, 10, 11, 12, \;\; \square, 15, 14, 13].$$

There are spaces at the row changes for clarity.

We further form classes of placements called *configurations*. Placements that can be reached from each other by moving the blank tile along the ordering path belong to the same configuration. Thus every configuration contains 16 placements, one for every position of the blank tile. We would write the configuration of the initial placement as follows:

$$C_{init} = [1, 2, 3, 4, 8, 7, 6, 5, 9, 10, 11, 12, 15, 14, 13].$$

Configurations have a parity which is determined by the number of misordered tiles. Let pos_i of a tile be its position in configuration C_i. In our example we could say $pos_{init}(8) = 5$.

For a given initial configuration C_0, a pair of numbered tiles x and y are misordered in configuration C_n if:

$$pos_0(x) < pos_0(y) \wedge pos_n(x) > pos_n(y).$$

Applied to our example we can say that C_{init} is even with 0 misorderings and that the configuration of the 14-15 puzzle's goal state is odd with 1 misordering.

Lastly, we characterize the effect of moving the blank tile on the puzzle's configuration by *permutations*. By definition, configurations do not change when moving the blank tile along the ordering path. Given this, there remain 9 possible moves, and their inverses, by which the blank tile can affect the puzzle's configuration. They are shown in Figure 2.4.

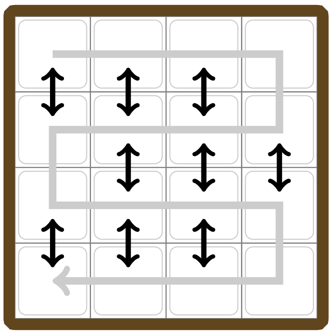


Figure 2.4: All moves that affect the puzzle's configuration.

In order to characterize these moves, we need a notion of parity for permutations. A permutation is even when it changes the number of misorderings by an even number. Two examples of this are illustrated in Figure 2.5.

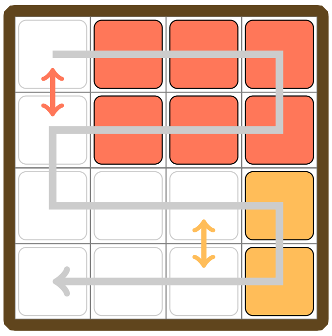


Figure 2.5: Two moves, each affecting an even number of orderings.

Taking a closer look, we can understand that the 9 moves that affect the puzzle's configuration all induce even permutations. Even permutations always preserve the parity of a configuration they are applied to. Archer (1999) further shows that all placements belonging to configurations of equal parity can reach each other.

Thus we have our desired result: because the initial and goal placement of the 14-15 puzzle belong to configurations of different parity, there is no sequence of moves that connects the two. The number of misordered tiles in a placement's corresponding configuration describes a function f with the properties described at the start of this section. In the following we introduce F_2, a useful mathematical concept for representing parity arguments.

$+$	0	1
0	0	1
1	1	0

$-$	0	1
0	0	1
1	1	0

$\times$	0	1
0	0	0
1	0	1

$\div$	0	1
0	–	0
1	–	1

Table 2.1: Arithmetic operations in F_2. Division is to be read *row* $\div$ *column*.

2.4 F_2

Fields are a fundamental mathematical construct and are used in many areas of mathematics and computer science. A field is a set over which the four basic operations addition, multiplication, subtraction and division (except by 0) are defined.

Common infinite fields are, for example, the real numbers $\mathbb{R}$, or the complex numbers $\mathbb{C}$. F_2 is the smallest finite field.

Definition 13 (Field F_2)**.** The field F_2 is the finite field with two elements. We call these elements 0 and 1. In F_2, addition and subtraction are equivalent to a logical XOR operation, multiplication is equivalent to a logical AND operation, and division is the identity function.

Table 2.1 illustrates the possible operations. As can be seen in the table, the additive and multiplicative identities are 0 and 1 respectively.

We have now seen parity arguments and the field F_2, which seems well suited for expressing such arguments. The next section will combine these concepts to form the central tool for our approach.

2.5 Potential Functions

In order to understand potential functions, we must first define *features*.

Definition 14 (Feature)**.** A feature f of planning task Π is a conjunction of atoms. The number of atoms is the *size* or *dimension* of the feature. A feature f is *true* in a state s (written $s \models f$) if all its atoms hold in s.

For convenience, we will write $vars(f)$ to refer to the set of variables mentioned in the atoms of f.

Potential functions were first proposed by Pommerening et al. (2015) as a family of heuristic functions. The idea of potential functions is to assign a *potential* (numerical value) to features. The result of the potential function is a linear combination of its features, weighted by their respective potentials. Usually potential functions are defined over the real numbers.

Definition 15 (Potential Function)**.** A potential function φ over planning task Π and features $\mathcal{F}$ is a linear combination of weighted features. The weights are determined by a *weight function* $w : \mathcal{F} \mapsto \mathbb{R}$. The potential value of state s according to the potential function φ for weight function w is:

$$\varphi(s) = \sum_{f \in \mathcal{F}} w(f)\ [s \models f].$$

A potential function's dimension is equal to the size of its largest feature.

Potential functions are a very general approach, and not all functions are useful. Constraints can be formulated to restrict the possible functions to ones with desirable properties. When using potential functions as heuristics, it is for example interesting to be able to limit the space of functions to ones that guarantee admissibility. Pommerening et al. (2015) found that efficient algorithms exist to discover goal-aware and consistent, and thereby admissible, potential heuristics of degree 1. The same result for functions of degree 2 was described by Pommerening et al. (2017), who further provide upper bounds on the number of required constraints and their size. For potential functions of degree ≥ 3 they state that no tractable algorithms exist.

With constraints in place to ensure sensible heuristics, an optimization function has to be chosen. The resulting optimization problem can be solved using linear programming which returns a potential heuristic. The choice of optimization function has a significant impact on the final heuristic. Pommerening et al. (2015) maximize the heuristic value of the initial state in their introductory implementation of potential heuristics. In further research, Seipp et al. (2015) evaluate various alternative optimization functions.

Besides heuristics, potential functions can also be adapted to unsolvability detection, where again, constraints have to be found to guarantee useful functions. A second important aspect is the choice of feature set. The simplest option is to use single atoms as features, resulting in potential functions of degree 1. Allowing features of size 2 expands the feature set and with it increases the expressive power of the resulting potential function. The trade off lies in the more complex constraints needed to ensure beneficial properties, and larger optimization problems.

As for our case, we want to encode parity arguments that can prove unsolvability. To do that, we need two elements to capture the conceptual even- and oddness of parity. We therefore restrict potential functions to F_2, which provides the elements we need and simultaneously simplifies theoretical and practical aspects of our approach.

Definition 16 (Potential Function in F_2)**.** A potential function φ in F_2 is a potential function with a weight function $w : \mathcal{F} \mapsto \mathrm{F}_2$ and a resulting potential function:

$$\varphi(s) = \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f].$$

Armed with all the necessary tools, concepts and definitions, we can now turn our focus to the theoretical aspects of this work.

3 Unsolvability Constraints in F_2

This chapter discusses the theoretical results of this thesis. The aim is to provide a set of constraints that describes potential functions in F_2 which encode parity arguments. The existence of such a potential function for a planning task should prove the task to be unsolvable. We first introduce the concept of separating functions and their connections to unsolvability. We then show how separating functions can be represented by potential functions in F_2. Lastly we provide practical constraints for up to two-dimensional features and how to represent them efficiently.

3.1 Separating Functions and Unsolvability

In this section we will introduce the concept of *separating functions* as a way to prove unsolvability.

The following proofs are based on Pommerening (2017). Statements are presented for planning tasks in TNF but can be extended to general SAS$^+$ tasks using the polynomial transformation by Pommerening and Helmert (2015).

Proposition 1. *A TNF planning task Π is unsolvable if there exists an s_0-invariant I that does not hold in the goal state.*

Proof. Invariant I holds in all states reachable from the initial state. Because I does not hold in the goal, it is not reachable and Π is therefore unsolvable. □

We can define such invariants using potential functions. The idea is to make the function separate the initial and goal state by making a transition between the two impossible. Such functions are called separating functions and are usually defined as a potential function φ over $\mathbb{R}$ with the following properties:

$$\begin{aligned} \varphi(s_0) &> \varphi(s_*) \\ \varphi(s) &\leq \varphi(s') && \text{for all transitions } s \xrightarrow{o} s'. \end{aligned}$$

The property $\varphi(s) \geq c$ is now a weak s_0-invariant proving unsolvability for any constant c with $\varphi(s_0) \geq c > \varphi(s_*)$. This argument can also be stated in terms of monotonically falling, instead of rising, potential values.

Separating functions of this kind have been described by Pommerening (2017) in the context of dead-end and, in extension, unsolvability detection. We will refer to this kind of potential function as *monotonic* or *numerical* separating functions.

For our purposes, a slightly different notion of separating functions is appropriate. We adapt our definition by generalizing the first condition to an inequality, and specializing the second condition by demanding potential values to be constant, as opposed to monotonically changing.

Definition 17 (Separating Function)**.** A potential function φ over a TNF planning task Π with initial state s_0 and goal state s_* is called a separating function if it satisfies the following constraints:

$$\begin{aligned} \varphi(s_0) &\neq \varphi(s_*) \\ 0 &= \varphi(s) - \varphi(s') && \text{for all transitions } s \xrightarrow{o} s'. \end{aligned}$$

In ambiguous contexts we may refer to our definition of separating functions as *parity functions.* We can now tie this notion of separating functions to unsolvability.

Proposition 2. *A TNF planning task* Π *is unsolvable if there exists a separating function* φ *over* Π*.*

Proof. A separating function cannot exist for a solvable task. Assuming such a function existed, any valid plan for the task would either contain a transition between states of unequal potential, or initial and goal state would have equal potential. Either statement would contradict one of the necessary conditions for a separating function to exist. Therefore separating functions can only exist for unsolvable tasks. □

3.2 Moving to F_2

While we could describe separating functions over any field, our specific application allows us to restrict ourselves to the smallest field F_2. Due to the binary nature of parity, it is sufficient for the separating function to distinguish two values. We can achieve this in a mathematically sound way by operating on the field F_2. The function value of a potential function is the linear combination of features and weights. Features are not affected by our choice of working within F_2, as they are solely determined by the dimensionality of the features. Weights on the other hand are now also restricted to the values of 0 or 1. The linear combination finally also uses the addition and multiplication operations defined by F_2 which become logical XOR and logical AND respectively.

With our scope set, we first translate the conditions for separating functions from the terms used in the definition, to a representation in F_2:

$$\begin{aligned} \varphi(s_0) &= \varphi(s_*) \oplus 1 \\ 0 &= \varphi(s) \oplus \varphi(s') && \text{for all transitions } s \xrightarrow{o} s'. \end{aligned}$$

This is valid because $a = b \oplus 1$ is equivalent to $a \neq b$ for $a, b \in \{0, 1\}$, and because addition and subtraction can both be seen as an XOR operation.

In a next step we plug the definition of potential functions in F_2 into the conditions above to get the full constraints:

$$\bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_0 \models f] = \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_* \models f] \oplus 1 \tag{1}$$

$$0 = \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f] \oplus \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s' \models f] \qquad \text{for all transitions } s \xrightarrow{o} s'. \tag{2}$$

Constraint 1 only depends on initial and goal state, and can therefore be expressed efficiently as a single constraint for any set of features. The same cannot be said about Constraint 2, as it implies a constraint for every state transition, which is not feasible in general. Starting with general transformation of the constraint, we will subsequently show how a compact representation is still possible for one- and two-dimensional feature sets.

We first syntactically simplify Condition 2:

$$\begin{aligned} 0 &= \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f] \oplus \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s' \models f] && \text{for all transitions } s \xrightarrow{o} s' \\ &= \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f] \oplus w(f) \wedge [s' \models f] && \text{for all transitions } s \xrightarrow{o} s' \\ &= \bigoplus_{f \in \mathcal{F}} w(f) \wedge ([s \models f] \oplus [s' \models f]) && \text{for all transitions } s \xrightarrow{o} s'. \end{aligned} \tag{3}$$

So far we have summed over all features in $\mathcal{F}$. We can now restrict that set without changing the sum. Specifically we can ignore all features that do not contain any variables mentioned in operator o. Formally, we can define this set as:

$$\mathcal{F}_{\bar{o}} = \{f \mid f \in \mathcal{F} \wedge \nexists V \in \mathit{vars}(f) : V \in \mathit{vars}(o)\}.$$

Operator o only affects variables mentioned in its effects and it can therefore never change the evaluation of features in $\mathcal{F}_{\bar{o}}$ over any transition $s \xrightarrow{o} s'$. More formally we can say that $[s \models f]$ iff $[s' \models f]$ for all features in $\mathcal{F}_{\bar{o}}$. Applying this observation to Equation 3, we can see that the right-hand conjunct equates to 0 for all features in $\mathcal{F}_{\bar{o}}$ because XOR is self-inverse: $1 \oplus 1 = 0$ and $0 \oplus 0 = 0$. Consequently, the conjunction evaluates to 0, which is XOR's identity element and therefore has no impact on the final result. We have hereby shown that all features in $\mathcal{F}_{\bar{o}}$ can be ignored, and only the complementary set of features with respect to $\mathcal{F}$ have to be considered. We can formally define this set as follows:

$$\mathcal{F}_o = \mathcal{F} \setminus \mathcal{F}_{\bar{o}} = \{f \mid f \in \mathcal{F} \wedge \exists V \in \mathit{vars}(f) : V \in \mathit{vars}(o)\}.$$

With this knowledge we can further simplify Equation 3 by only summing over features in $\mathcal{F}_o$:

$$0 = \bigoplus_{f \in \mathcal{F}_o} w(f) \wedge ([s \models f] \oplus [s' \models f]) \qquad \text{for all transitions } s \xrightarrow{o} s'. \tag{4}$$

For the general case of any-dimensional features, no more simplifications can be made. In further discussions we at times use $\Delta_o(f)$ to refer to the right-hand conjunct. We will now turn our focus to one- and two-dimensional potential functions specifically, and show how Constraint 2 can be efficiently expressed for such functions.

3.3 One-dimensional Potential Functions

One-dimensional potential functions are restricted to features of size one. Throughout this section, the base set of features $\mathcal{F}$ refers to the set of one-dimensional features. We can observe that, for this set of features, Equation 4 no longer requires full states s and s'. Instead, their references can be replaced by the partial states $pre(o)$ and $e\!f\!f(o)$ respectively. This is possible because the variables in $vars(f)$ for all $f \in \mathcal{F}_o$ are contained in $vars(o)$, which means, since we are only considering TNF, that all $v \in vars(o)$ occur in both $pre(o)$ and $e\!f\!f(o)$. Operator o therefore fully describes the value of features $\mathcal{F}_o$ in all states s and s' for which a transition $s \xrightarrow{o} s'$ exists. We can write Equation 4 as:

$$0 = \bigoplus_{f \in \mathcal{F}_o} w(f) \wedge ([pre(o) \models f] \oplus [e\!f\!f(o) \models f]) \qquad \text{for all operators } o \in \mathcal{O}. \tag{5}$$

Thus we have a compact representation for both Conditions 1 and 2. These equations form a sufficient condition for the existence of a one-dimensional separating function. Let us gather the relevant equations and cast a theorem based on our findings.

$$\bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_0 \models f] = \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_* \models f] \oplus 1 \tag{1}$$

$$0 = \bigoplus_{f \in \mathcal{F}_o} w(f) \wedge ([pre(o) \models f] \oplus [e\!f\!f(o) \models f]) \qquad \text{for all operators } o \in \mathcal{O} \tag{5}$$

Theorem 1. *Given TNF planning task* Π *and feature set* $\mathcal{F} = \mathcal{A}$*, the potential function* $\bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f]$ *is a separating function for all weight functions* $w : \mathcal{F} \mapsto \mathrm{F}_2$ *that satisfy Equation 1 and 5.*

3.4 Two-dimensional Potential Functions

For two-dimensional potential functions we will only consider two-dimensional features. In order to still be able to represent separating functions that contain both one- and two-dimensional features, we explicitly allow features with two identical atoms. These features behave the same way as their one-dimensional counterparts and seamlessly integrate into our following discussion. On the other hand, we explicitly exclude features that include two atoms containing the same variable but different values. Such features cannot hold in any state and we can therefore disregard them. Throughout this section, the base set of features $\mathcal{F}$ refers to the set of two-dimensional features, minus the aforementioned exceptions.

The approach for the set of two-dimensional features is less straightforward than the one-dimensional case. It is now no longer true that all variables in $\mathcal{F}_o$ are mentioned in $vars(o)$. Instead, this is only true for a subset of $\mathcal{F}_o$. According to this observation we divide $\mathcal{F}$ into two subsets, context-independent and context-dependent features, and treat them separately:

$$\mathcal{F}_o = \mathcal{F}_o^{ind} \cup \mathcal{F}_o^{ctx}.$$

First we deal with the context-independent features which we define as follows:

$$\mathcal{F}_o^{ind} = \{f \mid f \in \mathcal{F} \wedge \forall V \in vars(f) : V \in vars(o)\}.$$

Features with two identical atoms must be in $\mathcal{F}_o^{ind}$, because, as per definition of $\mathcal{F}_o$, the variable of at least one atom must be mentioned in o and therefore, with two identical atoms, both must be mentioned. This means that one-dimensional features are handled within $\mathcal{F}_o^{ind}$, through their two-dimensional representation.

The evaluation of the features in $\mathcal{F}_o^{ind}$ for states s and s' of transition $s \xrightarrow{o} s'$ is fully determined by operator o through $pre(o)$ and $\mathit{eff}(o)$. Analogously to the one-dimensional case, we can apply Equation 5 to $\mathcal{F}_o^{ind}$ and get a constraint over all context-independent features:

$$C_o^{ind} = \bigoplus_{f \in \mathcal{F}_o^{ind}} w(f) \wedge ([pre(o) \models f] \oplus [\mathit{eff}(o) \models f]).$$

We can make this constraint more explicit by considering the possible results of the right-hand conjunct, namely $\Delta_o(f) = [pre(o) \models f] \oplus [\mathit{eff}(o) \models f]$. Whenever $\Delta_o(f)$ is 0, we know that the feature f does not need to be considered, because $\Delta_o(f)$ makes the conjunction 0 and therefore has no impact on the value of C_o^{ind}. In the following we will strip down the definition of C_o^{ind} to only include significant features.

To aid this discussion we divide the atoms $\mathcal{A}_o = \bigcup_{V \in vars(o)} V$ into two sets:

$$\begin{aligned} \mathit{flips}_o &= pre(o) \bigtriangleup \mathit{eff}(o)^2 \\ \mathit{statics}_o &= \mathcal{A}_o \setminus \mathit{flips}_o. \end{aligned}$$

In words, flips_o captures all atoms that are either *consumed* or *produced* by o, and $\mathit{statics}_o$ captures all atoms in o that do not change. For a transition $s \xrightarrow{o} s'$, an atom a is produced if $[s \not\models a]$ and $[s' \models a]$, and conversely consumed if $[s \models a]$ and $[s' \not\models a]$.

We are looking at two-dimensional features of the form $f = a \wedge a'$ in $\mathcal{F}_o^{ind}$. By definition, both atoms in f are in $\mathcal{A}_o$ and consequently in either flips_o or $\mathit{statics}_o$. This allows us to divide $\mathcal{F}_o^{ind}$ into three subsets:

1. Both atoms are in $\mathit{statics}_o$.
2. Both atoms are in flips_o.
3. One atom is in flips_o, one atom is in $\mathit{statics}_o$.

We will analyze the three groups by constructing the truth tables over atoms a and a'. Atoms in $\mathit{statics}_o$ are either 0 or 1 and retain that value for $pre(o)$ and $\mathit{eff}(o)$. Atoms in flips_o are either produced (*prod*), meaning that they are 0 in $pre(o)$ and 1 in $\mathit{eff}(o)$, or consumed (*cons*), 1 in $pre(o)$ and 0 in $\mathit{eff}(o)$. The result of this analysis is shown in Table 3.1.

The first group yields the most straightforward result. All combinations result in 0 and we can therefore ignore this subset of features.

While the second group contains irrelevant features as well, namely features with one produced and one consumed atom, it also contains two relevant combinations. Features

[2] The symbol $\bigtriangleup$ refers to the symmetric set difference: $X \bigtriangleup Y = (X \cup Y) \setminus (X \cap Y)$.

$f = a \wedge a'$		$[pre(o) \models a \wedge pre(o) \models a']$ $\oplus$ $[eff(o) \models a \wedge eff(o) \models a']$
static	**static**	
0	0	$[0 \wedge 0] \oplus [0 \wedge 0] = 0$
1	1	$[1 \wedge 1] \oplus [1 \wedge 1] = 0$
0	1	$[0 \wedge 1] \oplus [0 \wedge 1] = 0$
1	0	$[1 \wedge 0] \oplus [1 \wedge 0] = 0$
flip	**flip**	
prod	*prod*	$[0 \wedge 0] \oplus [1 \wedge 1] = 1$
cons	*cons*	$[1 \wedge 1] \oplus [0 \wedge 0] = 1$
prod	*cons*	$[0 \wedge 1] \oplus [1 \wedge 0] = 0$
cons	*prod*	$[1 \wedge 0] \oplus [0 \wedge 1] = 0$
static	**flip**	
0	*prod*	$[0 \wedge 0] \oplus [0 \wedge 1] = 0$
0	*cons*	$[0 \wedge 1] \oplus [0 \wedge 0] = 0$
1	*prod*	$[1 \wedge 0] \oplus [1 \wedge 1] = 1$
1	*cons*	$[1 \wedge 1] \oplus [1 \wedge 0] = 1$

Table 3.1: Truth table for Δ_o of features $f \in \mathcal{F}_o^{ind}$.

where either both atoms are produced or both consumed return a $\Delta_o(f)$ of 1. We honor this fact by defining constraints over these features with:

$$\begin{aligned} \mathcal{F}^{ind}_{o,prod} &= \{f \mid f \in \mathcal{F}^{ind}_o \wedge \forall a \in f : a \in prod(o)\} \\ C^{ind}_{o,prod} &= \bigoplus_{f \in \mathcal{F}^{ind}_{o,prod}} w(f) \end{aligned} \tag{6}$$

and

$$\begin{aligned} \mathcal{F}^{ind}_{o,cons} &= \{f \mid f \in \mathcal{F}^{ind}_o \wedge \forall a \in f : a \in cons(o)\} \\ C^{ind}_{o,cons} &= \bigoplus_{f \in \mathcal{F}^{ind}_{o,cons}} w(f). \end{aligned} \tag{7}$$

We can observe a similar division in the third group where features are not relevant when the atom in $statics_o$ is 0 and become relevant when it is 1. Again, we cast this knowledge into a constraint for the relevant features:

$$\begin{aligned} \mathcal{F}^{ind}_{o,mix} &= \{f \mid f \in \mathcal{F}^{ind}_o \wedge \\ &\qquad \exists\, a \in f : a \in statics_o \wedge [pre(o) \models a] \wedge \\ &\qquad \exists\, a' \in f : a' \in flips_o\} \\ &= \{f \mid f \in \mathcal{F}^{ind}_o \wedge \\ &\qquad \exists\, a \in f : a \in \{pre(o) \cap eff(o)\} \wedge \\ &\qquad \exists\, a' \in f : a' \in flips_o\} \\ C^{ind}_{o,mix} &= \bigoplus_{f \in \mathcal{F}^{ind}_{o,mix}} w(f). \end{aligned} \tag{8}$$

Finally, we combine these constraints and create a new definition for C_o^{ind} that only contains all relevant features:

$$C_o^{ind} = C_{o,prod}^{ind} \oplus C_{o,cons}^{ind} \oplus C_{o,mix}^{ind}. \tag{9}$$

Next, we expand on the more complex, context-dependent features which are made up of the following set:

$$\mathcal{F}_o^{ctx} = \{f \mid f \in \mathcal{F} \wedge \exists V_o \in vars(f) : V_o \in vars(o) \wedge \exists V_{\bar{o}} \in vars(f) : V_{\bar{o}} \notin vars(o)\}.$$

Unlike the context-independent features, $f \in \mathcal{F}_o^{ctx}$ is not determined by o and instead depends on the states s and s'. With the goal to eliminate this state-dependence, let us consider a feature $f \in \mathcal{F}_o^{ctx}$ with $f = a \wedge \bar{a}$, where $var(a) \in vars(o)$ and $var(\bar{a}) \notin vars(o)$. We can understand that f can be described in terms of operator o by considering the truth value of $\bar{a}$ in state s. Either we have $[s \not\models \bar{a}]$ from which $[s \not\models f]$ and $[s' \not\models f]$ immediately follow, or we have $[s \models \bar{a}]$ (and therefore $[s' \models \bar{a}]$) in which case only o is needed to determine whether f holds in s and/or s'. Simply put, given the truth value of $\bar{a}$, the value of feature f is determined by o. Conceptually, the situation is similar to the third group of context-independent features, with the difference that here we do not know the value of $\bar{a}$.

The important thing to realize is that we do not have to care whether $\bar{a}$ holds in s and s' or not *if* we know that either case contributes the same to the final potential value. Because we know what features the value of $\bar{a}$ impacts, we can also make sure that, across all affected features, there is no difference between the two possible truth values of $\bar{a}$. With this guarantee, the potential value then becomes independent from s and s'. In the following paragraph we show how this result can be expressed through constraints.

Taking a closer look at $\mathcal{F}_o^{ctx}$, we find that a case distinction similar to $\mathcal{F}_o^{ind}$ is possible. Features $f = a \wedge \bar{a}$ of $\mathcal{F}_o^{ctx}$ can be assigned to one of two subsets:

1. Atom a is in $statics_o$.

2. Atom a is in $flips_o$.

The features of the first group are irrelevant for the same reason that the subset of $\mathcal{F}_o^{ind}$ with both atoms in $statics_o$ is irrelevant. The entry for features with both atoms in $statics_o$ in Table 3.1 shows that $\Delta_o(f)$ is always 0 if o changes neither of the atom's values.

The second group requires more special treatment. Here we cannot make any statements on the value of $\Delta_o(f)$ because it depends on $\bar{a}$. Instead, we will ensure that the features in this group contribute the same to the potential value, irrespective of $\bar{a}$.

Let us consider the variables $\bar{\mathcal{V}} = \mathcal{V} \setminus vars(o)$. The assignments of these variables are the possible atoms $\bar{a}$. We introduce the following constraint for every $V \in \bar{\mathcal{V}}$:

$$C_{o,V}^{ctx} = \bigoplus_{\substack{f = a \wedge \bar{a} \\ a \in flips_o}} w(f) \qquad \text{for all atoms } \bar{a} \in V. \tag{10}$$

This constraint states that, for a variable V, the sum of the weights of the features formed by combining an assignment of V with all atoms in flips_o, is equal for all assignments in V. In order to later construct the final constraint, we sum up all the constraints from above:

$$C_o^{ctx} = \bigoplus_{V \in \bar{\mathcal{V}}} C_{o,V}^{ctx}. \tag{11}$$

With this result we cover all features within the second group which correspond to the following set:

$$\mathcal{F}_{o,\mathit{flips}}^{ctx} = \{f \mid f \in \mathcal{F}_o^{ctx} \wedge \exists\, a \in f : a \in \mathit{flips}_o\}.$$

We have hereby handled all features in $\mathcal{F}_o^{ctx}$ and can compile the final constraint. By combining Equations 9 and 11 we get the following end result:

$$0 = C_o^{ind} \oplus C_o^{ctx} \qquad \text{for all operators } o \in \mathcal{O}. \tag{12}$$

This constraint serves as a compact representation of Condition 2 for the two-dimensional case. We can summarize our findings in the following theorem, after collecting the necessary equations.

$$\bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_0 \models f] = \bigoplus_{f \in \mathcal{F}} w(f) \wedge [s_* \models f] \oplus 1 \tag{1}$$

$$\begin{aligned} C_o^{ind} &= C_{o,prod}^{ind} \oplus C_{o,cons}^{ind} \oplus C_{o,mix}^{ind} && (9)\\ &= \bigoplus_{f \in \mathcal{F}_{o,prod}^{ind}} w(f) \oplus \bigoplus_{f \in \mathcal{F}_{o,cons}^{ind}} w(f) \oplus \bigoplus_{f \in \mathcal{F}_{o,mix}^{ind}} w(f) && \text{(using 6, 7, 8)} \end{aligned}$$

$$\begin{aligned} C_o^{ctx} &= \bigoplus_{V \in \bar{\mathcal{V}}} C_{o,V}^{ctx} && (11)\\ &= \bigoplus_{V \in \bar{\mathcal{V}}} \bigoplus_{\substack{f = a \wedge \bar{a} \\ a \in \mathit{flips}_o}} w(f) \qquad \text{for all atoms } \bar{a} \in V && \text{(using 10)} \end{aligned}$$

$$0 = C_o^{ind} \oplus C_o^{ctx} \qquad \text{for all operators } o \in \mathcal{O} \tag{12}$$

Theorem 2. *Given TNF planning task* Π *and feature set* $\mathcal{F} = \{a \wedge a' \mid a, a' \in \mathcal{A} \wedge (var(a) \neq var(a') \vee val(a) = val(a'))\}$*, the potential function* $\bigoplus_{f \in \mathcal{F}} w(f) \wedge [s \models f]$ *is a separating function for all weight functions* $w : \mathcal{F} \mapsto F_2$ *that satisfy Equation 1 and 12.*

Proof. Let $\psi(\mathcal{F}) = \bigoplus_{f \in \mathcal{F}} w(f) \wedge \Delta_o(f)$ be a function for the difference in potential value across transition $s \xrightarrow{o} s'$ for set of features $\mathcal{F}$.

$C_{o,V}^{ctx}$ combines the features in

$$\{f \mid f \in \mathcal{F}_o^{ctx} \wedge \exists\, a \in f : a \in V \wedge \exists\, a \in f : a \in \mathit{flips}_o\}$$

for a variable V. This variable holds exactly one value in state s and therefore $\Delta_o(f)$ is only 1 for a single feature in the above set in s. Because the weights of these features are bound to be equal, considering $C_{o,V}^{ctx}$ is equivalent to considering them all:

$$C_{o,V}^{ctx} = \psi(\{f \mid f \in \mathcal{F}_o^{ctx} \wedge \exists\, a \in f : a \in V \wedge \exists\, a \in f : a \in \mathit{flips}_o\}).$$

By summing over all variables in $\bar{\mathcal{V}}$, as in C_o^{ctx}, the covered features extend to:

$$C_o^{ctx} = \psi(\{f \mid f \in \mathcal{F}_o^{ctx} \wedge \exists\, a \in f : a \in \mathit{flips}_o\}).$$

For features f in $\mathcal{F}_o^{ctx}$ without an atom in flips_o it holds that $\Delta_o(f) = 0$. They can thus be added without affecting the outcome:

$$= \psi(\{f \mid f \in \mathcal{F}_o^{ctx}\}).$$

In order to construct C_o^{ind} we start with $C_{o,prod}^{ind}$ and $C_{o,cons}^{ind}$:

$$\begin{aligned} C_{o,prod}^{ind} \oplus C_{o,cons}^{ind} = \psi(\{f \mid\ & f \in \mathcal{F}_o^{ind} \wedge \\ & (\forall\, a \in f : a \in \mathit{prod}(o) \vee \\ & \ \forall\, a \in f : a \in \mathit{cons}(o))\}). \end{aligned}$$

This can be generalized by including features with one produced and one consumed atom, because their $\Delta_o(f)$ is 0:

$$= \psi(\{f \mid f \in \mathcal{F}_o^{ind} \wedge \forall\, a \in f : a \in \mathit{flips}_o\}).$$

$C_{o,mix}^{ind}$ looks as follows:

$$\begin{aligned} C_{o,mix}^{ind} = \psi(\{f \mid f \in \mathcal{F}_o^{ind} \wedge & \\ & \exists\, a \in f : a \in \mathit{statics}_o \wedge [\mathit{pre}(o) \models a] \wedge \\ & \exists\, a' \in f : a' \in \mathit{flips}_o\}). \end{aligned}$$

For features in the above set where a does not hold in $\mathit{pre}(o)$, $\Delta_o(f)$ is 0 and they can be included safely:

$$\begin{aligned} = \psi(\{f \mid f \in \mathcal{F}_o^{ind} \wedge & \\ & \exists\, a \in f : a \in \mathit{statics}_o \wedge \\ & \exists\, a' \in f : a' \in \mathit{flips}_o\}). \end{aligned}$$

For all features f in $\mathcal{F}_o^{ind}$ where both atoms are in $\mathit{statics}_o$ it is true that $\Delta_o(f)$ is 0. Consequently, they can be added to the other context-independent constraints to cover all features in $\mathcal{F}_o^{ind}$:

$$\begin{aligned} C_o^{ind} &= C_{o,prod}^{ind} \oplus C_{o,cons}^{ind} \oplus C_{o,mix}^{ind} \\ &= \psi(\{f \mid f \in \mathcal{F}_o^{ind}\}). \end{aligned}$$

Finally, we combine the results:

$$\begin{aligned} 0 &= C_o^{ctx} \oplus C_o^{ind} \\ &= \psi(\{f \mid f \in \mathcal{F}_o^{ind}\}) \oplus \psi(\{f \mid f \in \mathcal{F}_o^{ctx}\}) \\ &= \psi(\{f \mid f \in \mathcal{F}_o\}) \\ &= \bigoplus_{f \in \mathcal{F}_o} w(f) \wedge \Delta_o(f). \end{aligned}$$

$\square$

3.5 Considering Mutual Exclusivity

While Theorem 2 correctly defines a separating function over operators $\mathcal{O}$, there is still potential for improvement. As has been the theme of much of the discussion, we can once again identify features that do not have to be considered. Specifically we look at the constraints $C^{ctx}_{o,V}$ where V is a variable not in $vars(o)$. According to Equation 10, we demand $C^{ctx}_{o,V}$ to be equal for all atoms in V. The main observation in this section is that this step can be skipped for atoms a^{mutex}_o that are mutually exclusive with $pre(o)$ or $eff(o)$.

Doing this has no effect on the potential value of any reachable state s, because there is no transition $s \xrightarrow{o} s'$ where $[s \models a^{mutex}_o]$ or $[s' \models a^{mutex}_o]$ holds. This means that any feature including a^{mutex}_o remains false across all reachable transitions over o. Consequently, it does not matter whether the contribution of features including atoms a^{mutex}_o is equal to the contribution of assignments of V that are possible.

In order to incorporate this knowledge into the constraints, we must first adjust our definition of separating functions. So far, we required the equality constraint to apply to all transitions of a planning task. Seeing that the statements about mutexes can only be made for reachable states, we must restrict separating functions to the same domain. At the core of the separating functions we now have a weak s_0-invariant, when before it was strong.

It is important to note that we are not concerned with how the mutex information is obtained for now – this will be treated in Section 4.3. Instead, we simply assume that valid mutex pairs according to our Definition 12 are provided.

Proposition 3. *A TNF planning task Π is unsolvable if there exists a separating function φ over the reachable transitions of Π.*

Proof. Assuming the proposition is false, a separating function φ over reachable transitions and a valid plan π for task Π must coexist. The existence of φ guarantees different potential values for the initial and goal state. As π connects the initial and goal state, it has to bridge this difference along its path. Because valid plans can only exist over reachable transitions, a change of potential value along π directly contradicts the existence of φ. □

A proof for a stronger version of the above statement was shown for Proposition 2. With the prerequisites in place, we can now remove the components of $C^{ctx}_{o,V}$ that consider atoms in a^{mutex}_o and are still guaranteed that a resulting weight function proves unsolvability. More formally, we change Equation 10, which is defined over variables $V \in \mathcal{V} \setminus vars(o)$, to the following:

$$C^{ctx}_{o,V} = \bigoplus_{\substack{f = a \wedge \bar{a} \\ a \in flips_o}} w(f) \qquad \text{for all atoms } \bar{a} \in \{V \setminus a^{mutex}_o\}. \tag{13}$$

Proposition 4. *Given a set of mutexes M, operator o and variable V, any atom a^{mutex}_o for which there exists a mutex $m = \neg(a^{mutex}_o \wedge a)$ with $m \in M, a \in pre(o) \cup eff(o)$ can be excluded from $C^{ctx}_{o,V}$ without affecting Theorem 2 over reachable transitions.*

Proof. $C^{ctx}_{o,V}$ must be equal for all possible assignments of V in order to provide a consistent contribution to the potential value. Leaving the atoms a^{mutex}_o, which are impossible to hold in any reachable transition, out of the equation still yields this consistent contribution across all reachable transitions. $\square$

We can finalize our theoretical findings in a modified version of Theorem 2. For reference we restate the relevant equations here:

$$\bigoplus_{f\in\mathcal{F}} w(f) \wedge [s_0 \models f] = \bigoplus_{f\in\mathcal{F}} w(f) \wedge [s_* \models f] \oplus 1 \tag{1}$$

$$\begin{aligned} C_o^{ind} &= C_{o,prod}^{ind} \oplus C_{o,cons}^{ind} \oplus C_{o,mix}^{ind} && (9)\\ &= \bigoplus_{f\in\mathcal{F}_{o,prod}^{ind}} w(f) \oplus \bigoplus_{f\in\mathcal{F}_{o,cons}^{ind}} w(f) \oplus \bigoplus_{f\in\mathcal{F}_{o,mix}^{ind}} w(f) && \text{(using 6, 7, 8)} \end{aligned}$$

$$\begin{aligned} C_o^{ctx} &= \bigoplus_{V\in\bar{\mathcal{V}}} C_{o,V}^{ctx} && (11)\\ &= \bigoplus_{V\in\bar{\mathcal{V}}} \bigoplus_{\substack{f=a\wedge\bar{a}\\ a\in \mathit{flips}_o}} w(f) && \text{for all atoms } \bar{a} \in \{V \setminus a_o^{mutex}\}\\ & && \text{(using 13)} \end{aligned}$$

$$0 = C_o^{ind} \oplus C_o^{ctx} \qquad \text{for all operators } o \in \mathcal{O}. \tag{12}$$

Theorem 3. *Given TNF planning task* Π *and feature set* $\mathcal{F} = \{a \wedge a' \mid a, a' \in \mathcal{A} \wedge (var(a) \neq var(a') \vee val(a) = val(a'))\}$*, the potential function* $\bigoplus_{f\in\mathcal{F}} w(f) \wedge [s \models f]$ *is a separating function over the reachable transitions of* Π *for all weight functions* $w : \mathcal{F} \mapsto \mathrm{F}_2$ *that satisfy Equation 1 and 12 enhanced with Equation 13.*

4 Algorithm

As a basis for the implementation and evaluation discussion, we first describe how the presented concepts can be applied as an algorithm to detect unsolvability. The procedure consists of two main steps: constructing constraints and checking if they are satisfiable.

4.1 Constraint Construction

We represent the constraints as a matrix. As we are working in F_2, all entries are binary. There is a row for every constraint, a column for every *constraint variable*, and an additional result column. A constraint variable exists for every one- and two-dimensional feature as well as for every context variable.

In a constraint row, the constraint variables that are relevant to that constraint are set to 1, and all others to 0. Which constraint variables are relevant for which constraint is the result of the theoretical discussion and thus explained in detail in Chapter 3. Nevertheless, we briefly consider the practical implications of the result here. We consider the simple 2×2 instance of the sliding tiles puzzle shown in Figure 4.1, so as to illustrate these implications firsthand. We represent this task by the encoding given in Figure 4.2. Instead of listing

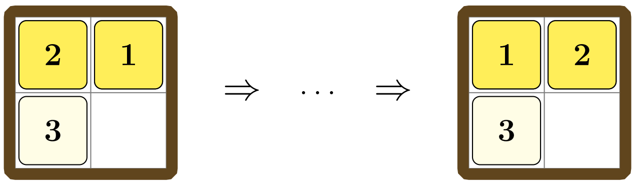


Figure 4.1: Simple, unsolvable sliding tile task.

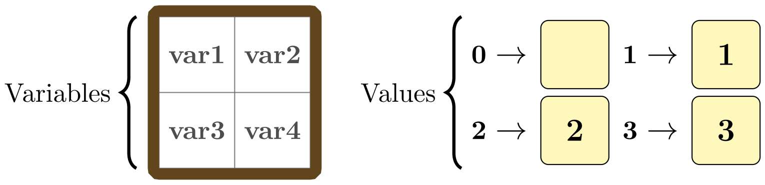


Figure 4.2: Example encoding for task in Figure 4.1.

all 122 rows that the algorithm generates for our example task, we will exemplify the three types of constraint rows: initial and goal row, operator rows, and context rows.

4.1.1 Initial and Goal Row

Initial and goal constraints induce one constraint row each. They set those constraint variables whose associated feature holds in the respective state. Additionally, the result column of the two rows must not be equal. For example, let the initial state's potential value be 0, and the goal state's 1. In the following we examine the goal condition of our example task given as a bit vector. The features are firstly ordered with ascending variables and secondly with ascending values.

$$\overbrace{\underbrace{0\,1\,0\,0}_{var_1}\ \underbrace{0\,0\,1\,0}_{var_2}\ \underbrace{0\,0\,0\,1}_{var_3}\ \underbrace{1\,0\,0\,0}_{var_4}}^{\text{One-dimensional features}}\overbrace{\dots 1 \dots 1 \dots \underbrace{1}_{\text{e.g. feature } \{var_1 \mapsto 1, var_4 \mapsto 0\}} \dots 1 \dots 1 \dots 1 \dots}^{\text{Two-dimensional features}}\overbrace{0 \dots 0}^{\text{Context variables}}\ \overbrace{1}^{\text{Result}}$$

There is exactly one active fact per variable, because the task is in TNF and the goal must describe a complete state. The six active two-dimensional features are the six combinations of two active one-dimensional features, namely $6 = \binom{4}{2}$. Context variables are only relevant for operator and context constraints and are therefore 0 here. Lastly the result column is 1, because we chose the result of the initial state to be 0.

Throughout this thesis we form initial and goal constraint rows as described, including implementation and subsequent evaluation. A more general approach would be to combine the rows into one by taking the bitwise XOR of the two rows. This would avoid having to arbitrarily assign 0 and 1 to initial and goal state, and would instead capture both possible assignments. We ran experiments with a combined initial and goal constraint row and could not find any differences besides minor variation in runtime.

4.1.2 Operator Row

Operator constraints have a larger impact on the constraint matrix, as their number scales with the size of the task. The relevance of the constraint variables referring to features is determined by whether the feature is contained in the context-independent constraint C_o^{ind} of operator o. Similarly, constraint variables referring to context variables are set if the context-dependent constraint $C_{o,V}^{ctx}$ exists for operator o and variable V. We consider the operator o that moves tile 3 from the bottom left to the bottom right position, formally $o = \langle\{var_3 \mapsto 3, var_4 \mapsto 0\}, \{var_3 \mapsto 0, var_4 \mapsto 3\}\rangle$, as an example.

$$\overbrace{\underbrace{0\,0\,0\,0}_{var_1}\ \underbrace{0\,0\,0\,0}_{var_2}\ \underbrace{1\,0\,0\,1}_{var_3}\ \underbrace{1\,0\,0\,1}_{var_4}}^{\text{One-dimensional features}}\overbrace{\dots\dots \underbrace{1}_{\{var_3 \mapsto 0, var_4 \mapsto 3\}} \dots\dots\dots\dots \underbrace{1}_{\{var_3 \mapsto 3, var_4 \mapsto 0\}} \dots\dots\dots}^{\text{Two-dimensional features}}\overbrace{\underbrace{1}_{C^{ctx}_{o,var_1}}\ \underbrace{1}_{C^{ctx}_{o,var_2}} \dots}^{\text{Context variables}}\ \overbrace{0}^{\text{Result}}$$

The operator does not affect variables var_1 and var_2 – their constraint variables are thus not relevant. Variables var_3 and var_4 contain two facts each that are mentioned in o – their constraint variables are set to 1. The pair of two-dimensional features shown are the only ones that appear in C_o^{ind} and consequently the only non-zero entries within two-dimensional features. The context variables look similar: two constraint variables are set for C^{ctx}_{o,var_1} and C^{ctx}_{o,var_2}, because var_1 and var_2 are not in $vars(o)$. These two context variables being set induces context rows that we cover in the following section. This operator row models the change in potential value across transitions using o. We want to ensure that this change is 0 and make the result column 0 accordingly.

4.1.3 Context Row

We just saw that context rows are induced by a context variable $C^{ctx}_{o,V}$ being set in an operator row. For every such case, $|dom(V)|$ context rows are constructed. We consider the row induced by C^{ctx}_{o,var_1} for $\{var_1 \mapsto 1\}$.

$$\overbrace{0\ldots0}^{\text{One-dimensional features}}\;\overbrace{\ldots\underbrace{1}_{\{var_1\mapsto1,var_3\mapsto0\}}\ldots\ldots\ldots\underbrace{1}_{\{var_1\mapsto1,var_3\mapsto3\}}\ldots\ldots\ldots\underbrace{1}_{\{var_1\mapsto1,var_4\mapsto0\}}\ldots\ldots\ldots\underbrace{1}_{\{var_1\mapsto1,var_4\mapsto3\}}\ldots\ldots}^{\text{Two-dimensional features}}\;\overbrace{\ldots\underbrace{1}_{C^{ctx}_{o,var_1}}\ldots}^{\substack{\text{Context}\\\text{variables}}}\;\overbrace{\;0\;}^{\text{Result}}$$

One-dimensional features are not relevant for the context row. Two-dimensional features are set for every combination of $\{var_1 \mapsto 1\}$ and one-dimensional feature active in the operator row for o. The context variable that spawned this row is the single active context entry. The result column must be 0 as we are encoding Equation 13, where $C^{ctx}_{o,V}$ is equal to a sum of two-dimensional features. In F_2, the equality $A = B$ can be expressed as $A \oplus B = 0$, which is what we are doing for this constraint.

The total size of the generated matrix for our example task is 122 rows by 161 columns. This is given the fact that we must only create constraint variables for those context variables $C^{ctx}_{o,V}$ where $V \notin vars(o)$. Looking at the presented example constraint rows, which are representative for this task, we can see that the matrix is rather sparse. This observation will be of interest in Chapter 5.

Having constructed the constraint matrix, the next step is to check if it is satisfiable.

4.2 Solving Constraints

The constraints matrix we have constructed describes an augmented matrix, that is, a matrix representation of a system of linear equations. Our aim now is to determine whether a solution exists for this system. If a solution does exist, we know that there exists a weight function w such that a potential function based on w describes a separating function, meaning that the task is unsolvable. Conversely, if no solution exists, no such weight function w exists and the task may or may not be solvable.

Gaussian elimination is a reliable algorithm to solve such systems and conveniently takes their augmented matrix as input. To keep our explanation simple, we assume a system with an equal number of variables and rows. In short, the procedure first brings the matrix into upper triangular form, where all entries below the diagonal are 0, and then into row echelon form, where all diagonal entries are 1. This is achieved using elementary row operations such as swapping rows, subtracting multiples of one row from another, and multiplying a row with a constant. Here, a simple example with rational numbers:

$$\left[\begin{array}{ccc|c} 2 & 0 & 1 & 1 \\ 4 & 2 & 7 & 3 \\ 0 & 2 & 4 & 0 \end{array}\right] = \ldots = \left[\begin{array}{ccc|c} 2 & 0 & 1 & 1 \\ 0 & 2 & 4 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right] = \ldots = \left[\begin{array}{ccc|c} 1 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right]$$

If Gaussian elimination finishes and there is a row where all variables are zero and the result non-zero, the system is inconsistent and no solution exists. Otherwise we can apply back substitution to get a solution.

There are two aspects of the algorithm we can simplify for our purposes. As we are working in F_2, it is easy to see that, for us, upper triangular and row echelon form are equivalent. Furthermore, we are not interested in finding a solution for the system, but merely in whether a solution exists or not. We can therefore terminate after seeing if the system is consistent and skip the back substitution step.

4.3 Mutex Generation

Blum and Furst (1995) first exploited mutex constraints in the context of planning. Ever since, mutexes have been studied in connection with various aspects of planning such as planning as SAT (Chen et al., 2007), regression planning (Alcázar et al., 2013) and even potential heuristics (Fišer et al., 2020).

With regards to mutex generation, a fundamental contribution has been made by Bonet and Geffner (2001). Their method is in turn based on the h^m heuristic proposed by Haslum and Geffner (2000). Due to a good trade-off between computational intensity and information content, the most common choice for m is 2. This makes h^2 the most frequently used method for generating mutexes. Other methods of invariant generation have been proposed, for example by Gerevini and Schubert (1998), and Rintanen (2000). For this thesis, we will utilize h^2 for its simplicity and efficiency.

5 Experimental Evaluation

In order to determine the practical use of the presented theory and resulting algorithm, we conducted a series of empirical experiments. First we briefly present the experimental setup, and then elaborate on optimizations, implementations, and results.

5.1 Setup

We implemented our algorithm in the Fast Downward[1] planning system by Helmert (2006). To enable SAS$^+$ to TNF task transformation, we included additional code by Florian Pommerening. The core functionality of Fast Downward and therefore our implementation is written in C++. To construct and run experiments we used Downward Lab by Seipp et al. (2017). Calculations were performed on Intel Xeon E5-2660 CPUs clocked at 2.2 GHz, provided by sciCORE[2] scientific computing center at University of Basel. Unless stated otherwise, resource limits were constant across all experiment runs with a time limit of 30 minutes and a memory limit of 3.5 GiB.

We tested our approach on a benchmark of unsolvable planning tasks by Eriksson (2019) containing 19 suitable domains with a total of 684 problems. We exclude the diagnosis domain because it contains conditional effects which our algorithm does not support.

In the following we will discuss how we implemented our algorithm, as presented in Chapter 4. We introduce the iterations we went through and quantify the performance differences. Additionally, we mention smaller scale optimizations that could affect performance.

5.2 Forget-Operator Fix

While testing an early version of our implementation on the sliding tile domain, we encountered an issue introduced by the transformation from SAS$^+$ to TNF task. When running the algorithm on an unsolvable instance, where the goal is a complete state but defined implicitly, the task transformation added unnecessary forget-operators.

[1] http://fast-downward.org
[2] https://scicore.unibas.ch

The constraints induced by these operators caused the constraint matrix to be unsatisfiable, even though the algorithm could successfully label an equivalent, explicit task as unsolvable.

We circumvent this issue through an additional check during the construction of operator constraints. We do not add a constraint for any forget-operator $o_{forget} = \langle\{V \mapsto v\}, \{V \mapsto \mathit{unknown}\}\rangle$ that satisfies the following conditions:

1. $\{V \mapsto \mathit{unknown}\}$ is a goal fact.
2. $\{V \mapsto v\}$ is mutex with at least one goal fact.
3. For every operator o with $\{V \mapsto \mathit{unknown}\} \in \mathit{pre}(o)$:
   $\{V \mapsto v\}$ is mutex with at least one other fact in $\mathit{pre}(o)$.

An operator o_{forget}, satisfying the three conditions above, cannot be part of a valid plan. Let us understand this by assuming that such a plan π containing o_{forget} exists. We can always rearrange the operator sequence of π so that o_{forget} only appears in front of operators o with $\{V \mapsto \mathit{unknown}\} \in \mathit{pre}(o)$ and/or as the last operator of the sequence. This is the case because V has a non-*unknown* value in the initial state and can retain that value until *unknown* is required explicitly.

We first consider the case of o_{forget} appearing in front of operator o. Forgetting the value of V directly before applying o implies that, in the non-TNF version of the task, V would still hold that value when applying o, just that we do not care about it in the TNF version. The contradiction arises with condition 3, which states that a state where $V \mapsto v$ holds, and o is applicable, is not reachable. Thus the sequence of these two operators cannot be part of a valid plan.

An identical argument can be made for the case of o_{forget} appearing as the last operator. According to condition 2, no state where $V \mapsto v$ and all goal conditions of the non-TNF task hold is reachable. Because all possible plans containing o_{forget} can be shown to be invalid, we can safely skip o_{forget} in the constraint construction step.

5.3 General Results

We now have all the prerequisites to implement a first version of our algorithm. Before we talk about the practical considerations of implementing our algorithm, we analyze the general results, using our initial implementation as the base line. Details about, and improvements upon this first implementation follow in Section 5.4.

Table 5.1 shows the results of this first implementation. The results suggest that parity arguments are only effective on a small number of domains. As expected, our algorithm performs well on the sliding-tiles domain, where all instances are successfully proven unsolvable via parity.

The second domain that shows success is the pegsol domain, where all tasks terminated. Our algorithm proves unsolvability for 22 out of 24 problems, and fails to do so for the remaining 2. It must be said that, during closer inspection of this domain, we realized that

	Proven by Parity	Proven by h^2	Not Proven	Out of Time	Out of Memory	Critical Error
3unsat (30)	–	–	**25**	–	5	–
bag-barman (20)	–	–	–	8	**12**	–
bag-gripper (25)	–	–	–	**14**	2	9
bag-transport (29)	–	**15**	1	3	10	–
bottleneck (25)	–	10	4	–	**11**	–
cave-diving (25)	–	1	8	3	**13**	–
chessboard-pebbling (23)	–	–	7	–	**16**	–
document-transfer (20)	–	2	2	5	**11**	–
mystery (9)	–	**9**	–	–	–	–
over-nomystery (24)	–	2	7	–	**15**	–
over-rovers (20)	–	3	5	–	**12**	–
over-tpp (30)	–	1	13	–	**16**	–
pegsol (24)	**22**	–	2	–	–	–
pegsol-row5 (15)	1	2	5	–	**7**	–
sliding-tiles (20)	**20**	–	–	–	–	–
tetris (20)	–	–	–	5	**15**	–
unsat-nomystery (150)	–	32	**93**	–	25	–
unsat-rovers (150)	–	62	8	–	**80**	–
unsat-tpp (25)	–	1	–	–	**24**	–
Sum (684)	43	140	180	38	**274**	9

Table 5.1: Outcomes of initial implementation we call full bitset.

our benchmark contains duplicate tasks. There are 12 unique instances with a duplicate each. As should be the case, our algorithm produces identical results for the duplicate pairs.

Peg solitaire, the board game modeled by the pegsol domain, is a well-studied puzzle. The (un)solvability of its various configurations and board shapes has been described alongside many more of its properties. We present some of these approaches as well as a detailed analysis of our own results for this domain in Chapter 6.

The pegsol-row5 domain offers some short-lived hope with its single successful task. Unfortunately this result cannot be attributed to our algorithm and is instead a product of the translator part of Fast Downward. While translating the original task, written in the Planning Domain Definition Language (PDDL) by Fox and Long (2003), into Fast Downward's internal representation, it is detected that the task's delete-relaxation, and therefore the task itself, is unsolvable. Knowing that, the translator replaces the actual task with a trivial, unsolvable dummy task which our algorithm then proves to be unsolvable.

There is another interesting observation involving the domains pegsol-row5 and the chessboard-pebbling. Both are well-known for the fact that their unsolvability can be proven using monotonic separating functions. Berlekamp et al. (2004) present such functions under the name *pagoda functions*, and use them to show that any finitely sized pegsol-row5 problem is unsolvable. They call the problem "The Solitaire Army"; it is also known as "Conway's Soldiers" after co-author John H. Conway. We will explore pagoda functions in more detail in Section 6.4. For the chessboard-pebbling problem, such a proof is shown, for example, by Chung et al. (1995). Despite the success of numerical separating functions, our algorithm is unable to show unsolvability for these domains. We interpret this as a symptom of the loss of generality incurred by our approach. This is supported by the fact that the unsolvability planner Aidos (and one of its components), which employs numerical separating functions, detects unsolvability for all tasks of these domains. We discuss the results of this comparison in Section 5.6.

We use mutexes generated by h^2 to relax constraints and have a better chance of finding parity functions. Even though these mutexes should only serve as a tool, they are capable of proving tasks to be unsolvable by themselves, by finding a mutex pair of which both atoms are contained in the goal. These cases are reflected in the "Proven by h^2" column of Table 5.1. In our initial analysis we falsely classified many of these cases as successes of our algorithm. We did not check if any of the found mutexes are contained in the goal and instead ran the entire procedure. For those tasks where the program terminated, it found a parity argument. It consisted of a single relevant, usually two-dimensional feature containing two goal atoms that h^2 found to be mutex. This is interesting because our algorithm does not treat the mutexes that already prove unsolvability themselves differently, but instead arrives at the same result guided by them and all other mutexes. Let us briefly examine a practical example.

Bottleneck is a domain where this case arises, that is easy to understand and visualize. A task consists of a series of connected locations and a set of persons. Every location is either active or not. Persons are in a location and can move to any connected, active location, deactivating it in the process. We consider the task pictured in Figure 5.1. We use green for active locations, red for inactive locations, and yellow for undetermined locations. After a

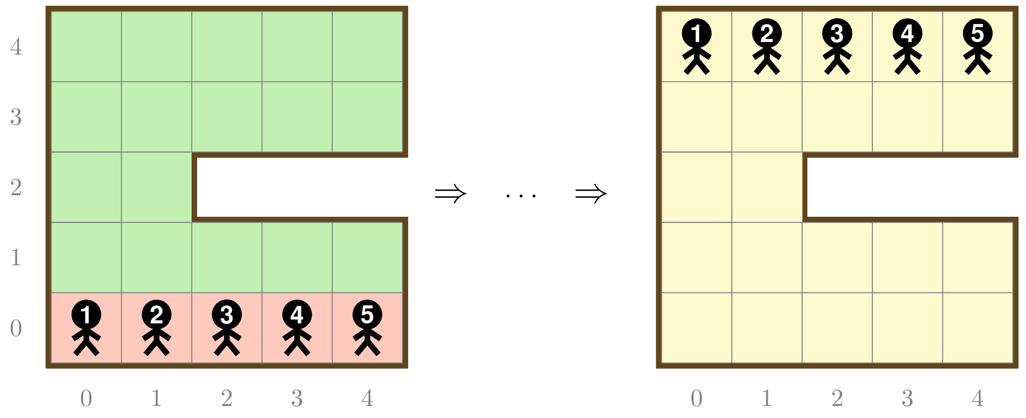


Figure 5.1: Initial and goal state of example instance.

series of moves from the task's initial state, we could for example arrive in the state shown in Figure 5.2.

We use the notation $per(5) \mapsto loc(0,4)$ to refer to the fact that person number 5 is in cell $(0,4)$, as is the case in the initial state. The only relevant feature for the separating function found by our algorithm is now $f = (per(2) \mapsto loc(4,1) \wedge per(3) \mapsto loc(4,2))$. We can see that f must hold in the goal and, after some trial-and-error, that there is indeed no state reachable from the initial state for which f holds. Finding this parity function is of course not necessary for an implementation. Feature f is also found to be mutex by h^2, and the program can terminate after mutex generation when checking if the goal contains any of them.

Our algorithm performs exceptionally badly on the domains bag-barman, bag-gripper,

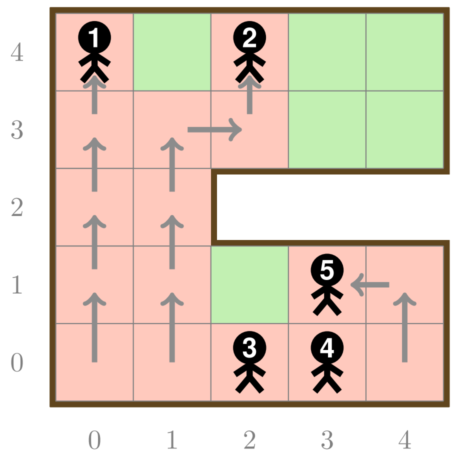


Figure 5.2: State after some moves from initial state (Figure 5.1).

	Number of Operators
3unsat	189
bag-barman	**12 822**
bag-gripper	**187 239**
bag-transport	3 388
bottleneck	602
cave-diving	2 249
chessboard-pebbling	579
document-transfer	3 991
mystery	2 918
over-nomystery	5 336
over-rovers	2 867
over-tpp	3 465
pegsol	76
pegsol-row5	441
sliding-tiles	278
tetris	**8 146**
unsat-nomystery	1 781
unsat-rovers	2 385
unsat-tpp	**15 110**

Table 5.2: Geometric mean over number of operators per domain. Excluded are bag-gripper tasks with critical error, and first task of pegsol-row5 with zero operators. The four largest results are in bold.

tetris, and unsat-tpp, where it does not terminate for any task. The single exception is one task of unsat-tpp where h^2 proves unsolvability. A possible explanation for this is the number of operators these tasks contain. Table 5.2 shows an overview for the number of operators in all domains. As indicated in bold, the four domains in question do indeed have significantly more operators than the other domains. The bag-gripper domain is the worst offender by an order of magnitude. The actual gap would presumably be even larger, because 9 of the biggest instances cause an error in the translator and we could not determine how many operators they contain. The fault in the translator may in fact also stem from the unusually high number of operators.

This concludes our implementation-independent analysis and we will now discuss the issues we faced and solutions we found while putting the algorithm into practice.

5.4 Implementations

The results presented in the previous section were produced by our first implementation. In this section we will discuss the important decisions we made during programming, how they affected performance, and what adjustments to those choices led to improvements. There are three implementations we analyze and compare.

5.4.1 Full Bitset

A crucial choice we have to make when implementing our approach, is what data type to use to store the constraint matrix. This is of importance because it is the largest object in our program with the biggest potential for memory usage, as well as the object operated on by Gaussian elimination, the most time consuming part of our algorithm. This choice therefore has implications for both limiting resources.

To store the collection of rows, we use a simple `std::vector`. Offering constant time element access and a way to reorder elements, it serves our purpose well. A container to store the rows themselves must fulfill more strict requirements. It namely needs to store a series of bits efficiently, allow quick access of a single bit, and offer a quick way to perform bitwise XOR over two rows. A `std::vector<bool>` for example satisfies the first two requirements but fails to provide efficient bitwise operations. With `std::bitset`, the standard library contains an alternative that would fit our three conditions, were it not for one caveat: its size has to be known at compile time. We can neither know the exact number of constraint variables nor define a sensible upper bound. The Boost library[3] defines `boost::dynamic_bitset` which has the same properties as `std::bitset` but its size is specified at runtime. An implementation of `dynamic_bitset` based on the Boost library exists within Fast Downward. We extend it with some functionality and use it to store the constraint rows.

As can be seen from the results on Table 5.1, the biggest problem with this implementation is its memory inefficiency, with “Out of Memory” being the most observed outcome. This is due to the size of the constraint matrix. Let us consider the largest problem of the bottleneck domain for which this implementation terminates. Its constraint matrix has a size of $176\,460 \times 96\,492$ which equates to a minimal size in memory of 2.0 GiB. The next larger bottleneck instance induces a matrix of size $265\,676 \times 102\,626$ with a minimal size in memory of 3.2 GiB. Together with other parts of the program, some overhead within `dynamic_bitset`, and an increase in memory usage during constraint construction, the program exceeds our memory limit of 3.5 GiB.

This example shows that storing the full constraints is not feasible for large tasks. In the following section we discuss our attempt to remedy this.

5.4.2 Sparse Set

One way to make matrices more memory-efficient is by only storing non-zero entries and their location, instead of all entries. In our case, we of course only have to store locations, because

[3] https://www.boost.org

all non-zero entries are 1. A location requires more memory than our 1 bit entries, which means that every non-zero entry introduces overhead. This way of representing matrices is therefore only beneficial for sufficiently sparse matrices. In other words, the memory saved by not storing zero entries must outweigh the added overhead for the remaining entries. As we briefly mentioned in Chapter 4, the constraint matrices we construct are quite sparse. Let us consider the same simple sliding tiles task from said chapter (see Figure 4.1). The total number of entries in the constraint matrix for that task is $122 \times 161 = 19\,642$ of which 693 are non-zero entries. This results in a sparsity of $1 - (693/19\,642) \approx 96.5\,\%$.

While we know that our matrices start out sparse, Gaussian elimination gives no guarantee that they remain sparse. Indeed, as shown in Figure 5.3 for our simple sliding tiles instance, we can observe a significant increase in the number of non-zero values during the execution of Gaussian elimination. Despite this potential problem, we implemented our

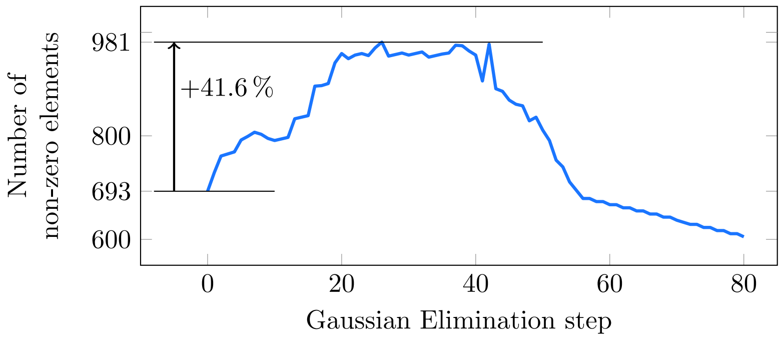


Figure 5.3: Number of non-zero entries in constraint matrix across Gaussian elimination execution for sliding tiles task (Figure 4.1).

algorithm using a sparse matrix representation to see how it behaves in practice.

A minor optimization we make is to dynamically choose the minimal fixed width integer type to store the indexes of non-zero entries. So instead of always using a 32 bit long `int`, we choose the smallest possible `unsigned int` with the length options being 8 bit, 16 bit, 32 bit and 64 bit. This is determined by the number of columns in the constraint matrix, which is known before construction begins.

With a sparse representation, we can no longer take advantage of efficient, bitwise XOR operations during Gaussian elimination. To achieve the same result we instead use `std::set_symmetric_difference`, which requires its input to be ordered and leads us to choosing a `std::vector` of `std::set` as the container. It allows us to directly construct the rows to be ordered and offers efficient lookup. Table 5.3 shows the results for the revised implementation.

The number of proven instances does not change. This is expected because the algorithm is identical and only the internal representation of the constraint matrix has changed.

We changed the representation with the aim to reduce memory usage, which we seem to have achieved, as the number of “Out of Memory” results is much lower compared to the full bitset implementation. Let us take a closer look and compare the peak memory usage of the two implementations. The results are shown in Figure 5.4.

A brief disclaimer before we begin the analysis: the numbers for all comparison plots

	Proven by Parity	Proven by h^2	Not Proven	Out of Time	Out of Memory	Critical Error
3unsat (30)	–	–	12 (−13)	**17**	1	–
bag-barman (20)	–	–	–	**16**	4	–
bag-gripper (25)	–	–	–	**14**	2	9
bag-transport (29)	–	**15**	1	6	7	–
bottleneck (25)	–	10	– (−4)	**15**	–	–
cave-diving (25)	–	1	8	**14**	2	–
chessboard-pebbling (23)	–	–	6 (−1)	8	**9**	–
document-transfer (20)	–	2	2	**16**	–	–
mystery (9)	–	**9**	–	–	–	–
over-nomystery (24)	–	2	1 (−6)	**18**	3	–
over-rovers (20)	–	3	4 (−1)	**13**	–	–
over-tpp (30)	–	1	6 (−7)	**14**	9	–
pegsol (24)	**22**	–	2	–	–	–
pegsol-row5 (15)	1	2	4 (−1)	**8**	–	–
sliding-tiles (20)	**20**	–	–	–	–	–
tetris (20)	–	–	–	**20**	–	–
unsat-nomystery (150)	–	32	**71** (−22)	47	–	–
unsat-rovers (150)	–	62	6 (−2)	**82**	–	–
unsat-tpp (25)	–	1	–	**19**	5	–
Sum (684)	43	140	123 (−57)	**327** (+289)	42 (−232)	9

Table 5.3: Outcomes of sparse set. Numbers in parentheses show difference to full bitset (Table 5.1).

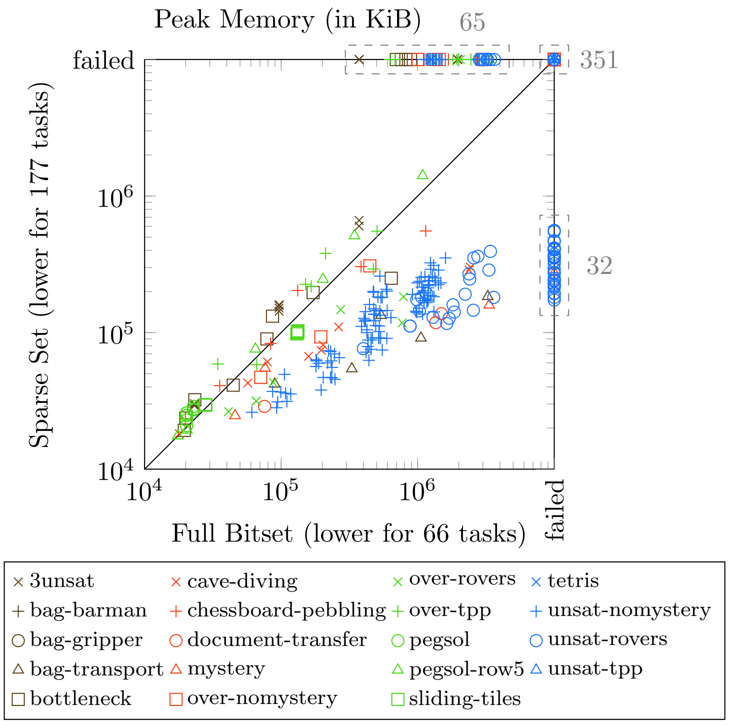


Figure 5.4: Peak memory comparison between full bitset and sparse set. Failed instances are excluded from “lower for”.

were generated by versions of the respective implementations that do *not* check for h^2 mutexes in the goal. Results for tasks where this is possible would be the same for all implementations. Instead we run the complete algorithm on all tasks in order to see the differences in performance more clearly.

There is a clear trend towards lower memory usage for the sparse set representation, especially with larger tasks. Despite this, there are still a considerable number of cases that confirm our earlier suspicions, where the sparsity is seemingly not high enough and the full bitset performs better. Another drawback of this second implementation stems from our choice of using `std::set` as the row container. While not required by the C++ standard, `std::set` is usually implemented as a self-balancing binary search tree. We use libstdc++ which indeed implements `std::set` as a red-black tree, where nodes need to store two pointers to their children on top of their key/data. This overhead can be significant in our case, considering that we store many small keys.

While we successfully tackled the symptom of high memory usage for the most part, we did not improve upon the first implementation. The program now terminates for 57 fewer tasks in total and instead exceeds the 30-minute time limit more often. We visualize the difference in total computation time in Figure 5.5. Except for a handful of

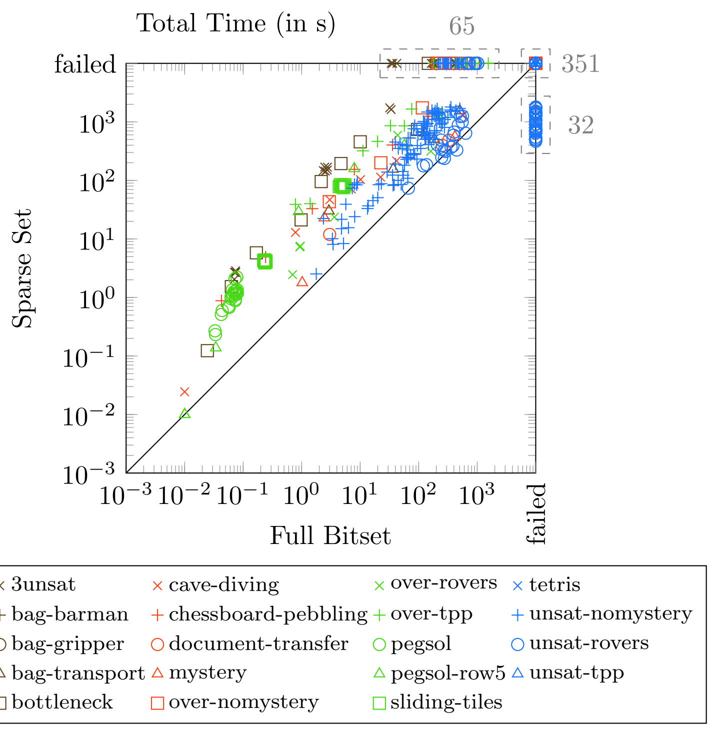


Figure 5.5: Total time comparison between full bitset and sparse set.

outliers, the full bitset version is consistently faster. Two factors that could possible lead to this behavior are less efficient XOR operations and cache-unfriendliness. While both `std::symmetric_set_difference` and bitwise XOR on `dynamic_bitset` have linear time complexity in the number of elements in the two sets, `dynamic_bitset` should have an advantage in practice as it can process 32 bit long chunks, instead of comparing single bits. The second potential issue is that `std::set` is not stored contiguously in memory, as opposed to `dynamic_bitset` which is a vector at its core. This can lead to significantly more cache misses during Gaussian elimination.

	Proven by Parity	Proven by h^2	Not Proven			Out of Time			Out of Memory			Critical Error
3unsat (30)	–	–	**25**		(+13)	5			–			–
bag-barman (20)	–	–	–			**20**			–			–
bag-gripper (25)	–	–	–			**16**			–			9
bag-transport (29)	–	**15**	2	(+1)	(+1)	5			7			–
bottleneck (25)	–	10	4		(+4)	**11**			–			–
cave-diving (25)	–	1	10	(+2)	(+2)	**14**			–			–
chessboard-pebbling (23)	–	–	9	(+2)	(+3)	**13**			1			–
document-transfer (20)	–	2	2			**16**			–			–
mystery (9)	–	**9**	–			–			–			–
over-nomystery (24)	–	2	9	(+2)	(+8)	**13**			–			–
over-rovers (20)	–	3	8	(+3)	(+4)	**9**			–			–
over-tpp (30)	–	1	13		(+7)	**16**			–			–
pegsol (24)	**22**	–	2			–			–			–
pegsol-row5 (15)	1	2	**6**	(+1)	(+2)	**6**			–			–
sliding-tiles (20)	**20**	–	–			–			–			–
tetris (20)	–	–	–			**20**			–			–
unsat-nomystery (150)	–	32	**101**	(+8)	(+30)	17			–			–
unsat-rovers (150)	–	**62**	40	(+32)	(+34)	48			–			–
unsat-tpp (25)	–	1	–			**24**			–			–
Sum (684)	43	140	231	(+51)	(+108)	**253**	(+215)	(−74)	8	(−266)	(−34)	9

Table 5.4: Outcomes of sparse vector. Numbers in parentheses show difference to full bitset (Table 5.1) on the left, and difference to sparse set (Table 5.3) on the right.

Sparse representation is a promising approach, yet it fails to outperform the full bitset. In a third iteration we try to remedy issues with sparse representation in order to surpass both previous approaches.

5.4.3 Sparse Vector

In the previous section we saw multiple issues that could be explained by our choice to store rows as `std::set`. For this third and last iteration we hope to improve the sparse approach by replacing `std::set` with `std::vector`. Our reasoning for choosing `std::set` initially was the fact that it is ordered, and offers lookup with time complexity $\mathcal{O}(\log n)$ for a set of length n. It turns out that we can achieve the same properties using `std::vector` with little effort, by simply sorting all rows after construction. This is only necessary once, because `std::symmetric_set_difference` preserves order and is the only function modifying rows after initial construction. The fact that the rows are sorted, further allows us to use `std::binary_search` to achieve equally efficient lookups to `std::set`.

A conceivably critical advantage of `std::vector` over `std::set` is it being contiguous in memory, which should make it more cache-friendly. Table 5.4 shows how well the theoretical advantages hold up in practice. To which the answer is: rather well. While there is still no change with regards to proven unsolvability, this time we did improve the implementation, which now terminates for more instances than both our previous attempts.

We again will look at measures of peak memory and time across all tasks. We compare against the stronger previous implementation for both categories. Let us first look at memory usage compared to the sparse set approach, shown in Figure 5.6. The vector version strongly dominates the set version, and the effect becomes more pronounced with larger instances. We assume that this is caused by the lack of overhead induced by `std::vector` as opposed to `std::set`.

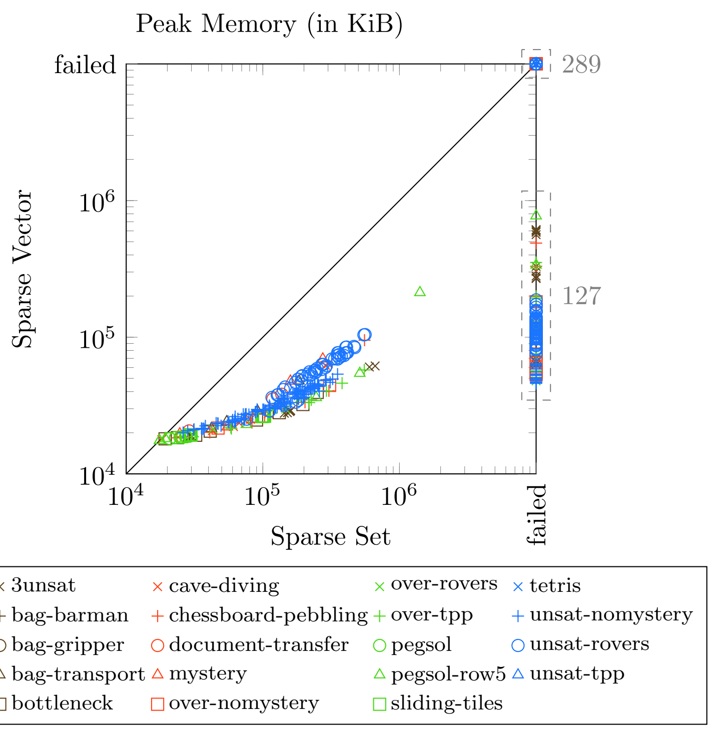


Figure 5.6: Peak memory comparison between sparse set and sparse vector.

Let us also consider computation time again, where the sparse set approach struggled the most. To better judge the time performance of sparse vector, we compare it against the fast bitset instead of the slow sparse set. See the resulting plot in Figure 5.7. Perhaps surprisingly, the vector implementation can match the full bitset very closely. It seems that bitset has the edge for smaller tasks, but larger tasks are spread more evenly or slightly fall into the favor of the sparse approach. We think that the better memory locality, and resulting cache-friendliness of `std::vector` could be an important contributing factor for the improved performance over `std::set`.

5.5 Sliding Tiles

Variations on the 15 puzzle have been our primary examples throughout this thesis. We have used them in theoretical discussion as well as for practical consideration. The instances in the sliding-tiles domain are comprised of ten 3×3 and ten 3×4 sized problems, which both the full bitset and sparse vector implementations can prove very quickly. We present a brief comparison in Table 5.5 under “sliding-tiles”. To better gauge the potential of our approach for proving unsolvability in the sliding-tiles domain, we test our algorithm’s performance on larger tasks. We will henceforth not consider the sparse set implementation anymore, as it performs significantly worse than its competitors.

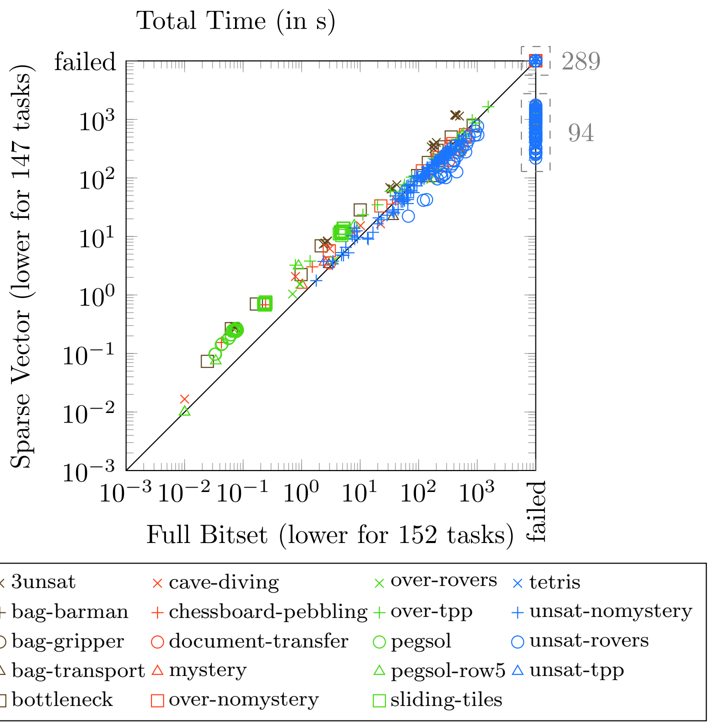


Figure 5.7: Total time comparison between full bitset and sparse vector. Failed instances are excluded from "lower for".

		Total Time (in s)	Peak Memory (in KiB)
sliding-tiles (geometric)	Full Bitset	**1.0**	105 019
	Sparse Set	18.0	96 195
	Sparse Vector	2.5	**59 931**
15 Puzzles (arithmetic)	Full Bitset	181.3	1 388 748
	Sparse Vector	**159.9**	**91 798**
24 Puzzles (arithmetic)	Sparse Vector	7 354.9	637 952

Table 5.5: Time and memory comparison across sliding-tiles tasks of increasing size. In parentheses the averaging method, either geometric or arithmetic mean.

5.5.1 15 Puzzle

We first consider tasks with a 4×4 board, whose size corresponds to the classic 15 puzzle. We test on 100 randomly generated problems, posed by Korf (1985). The problems are solvable, have varying initial states but share the same goal state. The goal state has the blank in the top left corner and tiles in ascending order, row-wise from top to bottom. To make the tasks unsolvable, we switched the positions of tile 1 and 2 in the goal so that the top row reads $\langle \square, 2, 1, 3 \rangle$, instead of $\langle \square, 1, 2, 3 \rangle$. This change should not affect the randomness of the instances. The results are shown in the "15 Puzzles" row in Table 5.5.

Note that we use arithmetic mean to sum up the numbers, instead of geometric as we used previously. We do this because all tasks are the same size here and, per algorithm, values are very similar. The memory usage of full bitset is in fact constant, because the dimensions of the constraint matrices are the same for all tasks.

Interestingly, sparse vector solves these larger problems faster than full bitset. A likely explanation for this is that sparse vector only operates on non-zero entries, while full bitset operates on full rows. Let us investigate how the sparsity of constraint matrices scales with larger sliding-tiles instances.

5.5.2 Sparsity Scaling

For this discussion, we can disregard the initial and goal constraint rows, because their impact is dominated by operator constraints. Although their length changes with task size, their number (2) remains constant, as opposed to operator constraints, which become both longer and more numerous with bigger boards. It is further sufficient to examine the case of a single operator row and the context rows it induces, because all operators in sliding-tiles share the same properties, such as the number of preconditions and effects, and behave identically with respect to sparsity. This means that the change in sparsity observed in an operator constraint row and the context rows it induces, can be generalized to describe the task as a whole.

Let us first look at how the number of constraint variables can be determined for a sliding-tile task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$ with domain $d = dom(V)$ for all $V \in \mathcal{V}$:

- Number of one-dimensional features: $|\mathcal{V}| \cdot |d|$
- Number of two-dimensional features: $\binom{|\mathcal{V}|}{2} \cdot |d|^2 = \frac{|\mathcal{V}|^2 - |\mathcal{V}|}{2} \cdot |d|^2$
- Number of context constraint variables: $|\mathcal{O}| \cdot (|\mathcal{V}| - 2)$

When expanding Π, while preserving its underlying idea, the smallest possible step we can take is to add a single cell containing a tile. This gives us the expanded task Π' with $\mathcal{V}' = \mathcal{V} \cup V_{new}$ and $d' = d \cup v_{new}$. Both adding a value to d, and adding a variable to $\mathcal{V}$ cause a polynomial, non-linear increase in the number of two-dimensional features. Similarly, the number of operators grows linearly with respect to $|\mathcal{V}|$ and through being multiplied by $(|\mathcal{V}| - 2)$, makes the increase in the number of context constraint variables non-linearly polynomial in $|\mathcal{V}|$. Therefore we can conclude that the total number of constraint variables grows polynomially, and faster than linear, in the size of Π.

The second component we must understand is the number of non-zero entries. To that end, we first examine the number of non-zero constraint variable values in an operator row for operator o:

- Number of set one-dimensional constraint variables: 4
  There are 4 atoms in $pre(o) \triangle eff(o)$.
- Number of set two-dimensional constraint variables: 2
  There is 1 pair of atoms being simultaneously produced, and 1 pair being simultaneously consumed.
- Number of set context constraint variables: $|\mathcal{V}| - 2$
  There are 2 variables in $vars(o)$.

	Sparsity (in %)
2×2	96.47
3×3	99.88
4×4	99.95

Table 5.6: Sparsity in sliding-tiles problems of increasing size.

We now know that, for operator rows, the number of non-zero entries grows linearly in the size of Π. The only missing piece is whether the same is true for all context rows induced by o. As with operator rows, it is sufficient to consider a single context row, because they all behave identically when considering the number of non-zero entries they contain. We examine a context row for Atom A in $V \notin \mathit{vars}(o)$ with $V \in \mathcal{V}$:

- Number of set one-dimensional constraint variables: 0
  These are not relevant for context variables.
- Number of set two-dimensional constraint variables: 4
  There are 4 atoms in $\mathit{pre}(o) \bigtriangleup \mathit{eff}(o)$, each paired with A.
- Number of set context constraint variables: 1
  The context variable for o and V.

Seeing that context rows contain a constant number of non-zero entries in the size of Π, we have considered all components necessary to understand how sparsity behaves.

Proposition 5. *Let the size of sliding-tiles planning task Π with domain $d = \mathit{dom}(V)$ for all $V \in \mathcal{V}$ be the number of its atoms $|\mathcal{A}|$. The sparsity of the constraint matrix M induced by Π increases monotonically with the size of Π.*

Proof. Increasing the number of atoms $|\mathcal{A}|$ must increase either $|\mathcal{V}|$, $|d|$, or both. The total number of constraint variables in M grows polynomially, and faster than linear, with increasing $|\mathcal{V}|$ or $|d|$. Meanwhile, the number of non-zero entries in M grows only linearly with respect to $|\mathcal{V}|$ and $|d|$. Therefore, the sparsity of M given by $1 - \frac{\#\text{non-zero entries}}{\#\text{constraint variables}}$ increases monotonically with the size of Π. $\square$

After establishing the theory, we now look at a sequence of sparsity values over increasing sliding-tiles tasks. Specifically, we consider the progression of square boards from 2×2 to 4×4, and measure the sparsity of the resulting constraint matrix before Gaussian elimination. The numbers are listed in Table 5.6. To put this sparsity discussion into perspective, it is important to realize that these results only hold for the initially constructed constraint matrix, and that standard Gaussian elimination can make the matrix significantly less sparse during computation. We have previously touched on this in Section 5.4.2, and specifically Figure 5.3.

5.5.3 24 Puzzle

We have seen that our algorithm, given the right implementation, can handle 15 puzzle problems within acceptable resource bounds. Further we explored the favorable scaling properties of the sliding-tiles domain for sparse matrix representation. These discoveries give us reason to wonder how we would fare on even larger boards. To test just that, we consider the next biggest square, 4×4, and arrive at the 24 puzzle.

This is a very significant step up in size as we go from the 15 puzzle's $16! \approx 2.1 \times 10^{13}$ states, to $25! \approx 1.5 \times 10^{25}$ for the 24 puzzle. Task sizes of this magnitude make our full bitset implementation infeasible. Our algorithm constructs a constraint matrix of size $1\,017\,602 \times 232\,286$ for 24 puzzle instances, which means a theoretical minimal memory requirement of 27.5 GiB for storing the full matrix. We therefore evaluate only the sparse vector implementation with the same memory limit as for all previous experiments, 3.5 GiB, and a generous time limit of 8 h.

As for the instances, we consider 50 problems posed by Korf and Felner (2002). We made the instances unsolvable in the same way as the 15 puzzle problems, by switching two tiles in the shared goal state. The results of the evaluation are shown in the "24 Puzzles" row on Table 5.5.

The sparse vector implementation of our algorithm solves all 50 instances with both time and memory to spare. The majority of times are within 10 min of the 2.0 h average runtime with a handful of outliers that finish just over 15 min faster than average. Inconsistencies arose with another experiment run with identical setup, where times fluctuated from 2.4 h down to 1.5 h. At present we assume that the variation in these results stem from the computer cluster we ran our experiments on. Memory on the other hand is stable and consistent across runs. The fact that memory usage increased by a factor of less than 10 with respect to 15 puzzles, while the number of states increases in the order of 10^{14}, supports our findings of growing sparsity with growing task size.

5.6 Aidos

Now that we have an overview of the performance, strengths, and weaknesses of our approach in practice, we will compare it to a state of the art planner built for unsolvability detection. For this comparison we have chosen Aidos, the winner of the Unsolvability IPC 2016. Aidos, developed by Seipp et al. (2016), is a collection of three portfolios of which we consider the one that performed best in the competition. As in their paper we call this configuration *Aidos 1*. Aidos 1 was constructed using a variety of techniques which culminate in three main components: resource detection using A* search, and breadth-first search using either dead-end pattern databases or dead-end potentials with two-dimensional features.

The last component is what makes Aidos interesting to compare our algorithm to. Dead-end potentials encode monotonic separation functions, which are a generalization of the separating functions our approach seeks to find, as we have seen in Section 3.1. Another parallel we can draw to our approach is the usage of h^2, which they do in a preprocessing step, based on ideas by Alcázar and Torralba (2015). Importantly, they use the mutexes found by h^2 exclusively to simplify the tasks, and do not consider them anymore thereafter.

With these observations in mind, we tested one Aidos portfolio and a modified version of the dead-end potential component on our benchmark set. Firstly, we ran the default portfolio Aidos 1, as it appeared in the Unsolvability IPC 2016. Secondly, we ran the dead-

	Parity Arguments	Dead-End Potentials	Aidos 1
3unsat (30)	–	–	**30**
bag-barman (20)	–	–	**12**
bag-gripper (25)	–	**15**	**15**
bag-transport (29)	–	22	**23**
bottleneck (25)	–	**25**	**25**
cave-diving (25)	–	2	**8**
chessboard-pebbling (23)	–	**23**	**23**
document-transfer (20)	–	10	**13**
mystery (9)	–	**9**	**9**
over-nomystery (24)	–	5	**14**
over-rovers (20)	–	5	**13**
over-tpp (30)	–	12	**26**
pegsol (24)	22	4	**24**
pegsol-row5 (15)	1	**15**	**15**
sliding-tiles (20)	**20**	–	10
tetris (20)	–	**20**	**20**
unsat-nomystery (150)	–	62	**149**
unsat-rovers (150)	–	76	**139**
unsat-tpp (25)	–	13	**25**
Sum (684)	43	318	**593**

Table 5.7: Number of tasks proven unsolvable by our approach, and Aidos .

end potential component, but limited it to only evaluate the initial state instead of searching, in order to see for what tasks it can find monotonic separating functions for the initial state. The results are summed up in Table 5.7. Unsurprisingly, Aidos 1 outperforms both our approach and the single dead-end potential configuration swimmingly. It is noteworthy though, that dead-end potentials, even with our additional restriction, keep up well and can match Aidos 1 for several domains. This suggests that monotonic separating functions are useful in various domains, especially when compared to parity functions. We previously mentioned that the unsolvability of pegsol-row5 and chessboard-pebbling is known to be provable using monotonic separating functions. Our experiment confirms that dead-end potentials are capable of detecting these proofs automatically.

The two domains where our approach finds success offer the most compelling point of comparison. The numbers for sliding-tiles confirm that our algorithm's strength in this domain remains significant with state of the art competition. Even beyond Aidos, the sliding-tiles tasks in our benchmark are the same 20 instances used in the Unsolvability IPC 2016. No planner in the competition was able to show more than 10 of these tasks to be unsolvable, with the majority of competitors showing 10 exactly[4]. We assume that they correspond to the half of problems that are 3×3 in size, whereas the bigger half is 3×4.

The fact that dead-end potentials fail for all sliding-tiles problems warrants an explanation. Because their underlying concepts are more powerful than parity, we hypothesize that the crucial difference lies in the consideration of mutexes. This is supported by the fact that our algorithm also fails for all sliding-tiles tasks when we do not consider h^2 mutexes in the constraint construction step.

Unlike the strong performance of Aidos 1, the pegsol results for dead-end potentials

[4] Competition domains and results from https://github.com/AI-Planning/unsolve-ipc-2016

are surprising. Even though dead-end potentials employ monotonic separating functions and two-dimensional features, they do not successfully detect unsolvability for 18 cases where parity arguments do. We suspect that these arguments require the cyclic `modulo 2` property inherent to parity arguments, which cannot be expressed in real-valued potential functions. In the next chapter we take an in-depth look at unsolvability proofs in the pegsol domain.

6 Peg Solitaire Case Study

In this chapter we examine parity arguments our algorithm finds for peg solitaire problems. We draw parallels to known results and consider the application of numerical arguments.

Peg solitaire is a single player board game that has existed for over 300 years. It consists of a board with holes, and pegs that can occupy the holes. Generally, a problem starts out with a number of pegs arranged on the board. The aim then is to remove pegs and be left with the remaining pegs in a predefined end configuration. Most often, only a single peg should remain. Pegs are removed by a jumping move, where one of two adjacent pegs leaps over the other and lands in the neighboring, unoccupied hole. The stationary peg is then removed. Figure 6.1 illustrates such a move. Jumping is allowed horizontally and vertically,

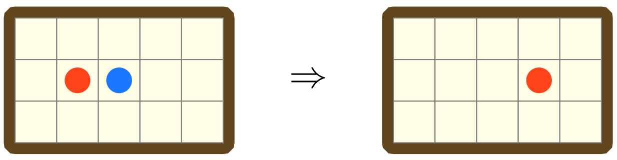


Figure 6.1: Peg solitaire move, red jumps over blue.

but diagonal jumps are not permitted.

We consider the most popular board variant, which has 33 holes and is called the "English" board. We draw the board as a grid where empty cells are free holes and cells containing circles are holes occupied by a peg. Figure 6.2 shows the classic *central game* where the initially free hole must be left with the last remaining peg in the end.

In the following sections, we will take a look at unsolvable peg solitaire problems posed by the pegsol domain of our benchmark collection. In particular, we want to understand the parity arguments generated by our algorithm and the reasons why it fails for some tasks. While our algorithm does not normally calculate a solution to the constraint matrix, we did so for some pegsol instances and analyze the results here. A solution for the constraint matrix is a vector containing the weights for all features. These weights form a weight function from which we can formulate a potential function, which in turn encodes a parity argument for the task at hand. We will study the nature of the constructed parity arguments by examining the solution vectors directly.

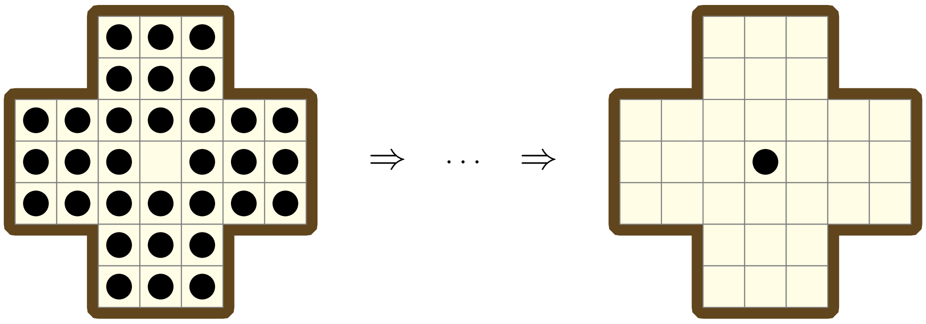


Figure 6.2: Central game problem on the English board.

6.1 TASK1

Let us consider the first instance of the pegsol domain and call it TASK1. Its initial and goal states are pictured in Figure 6.3. Our algorithm successfully proves this problem to be

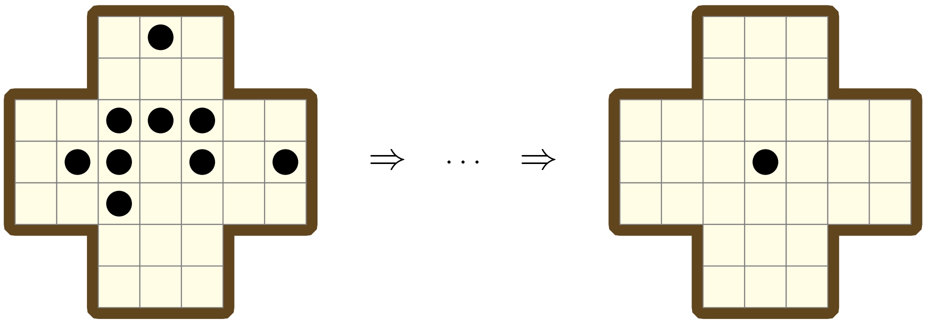


Figure 6.3: TASK1, first instance of pegsol domain.

unsolvable. The resulting solution vector reveals a first interesting point: no two-dimensional features are used for the parity argument. Surprisingly, one-dimensional features suffice for proving TASK1 unsolvable. To understand the argument, we first visualize the solution vector by marking the relevant board locations. There are two atoms for every location: one for the hole being free, and one for it being occupied. We color locations where the free-atom is set in the solution vector green, and those where the occupied-atom is set red. The result is shown in Figure 6.4. Considering that the initial state is relatively chaotic,

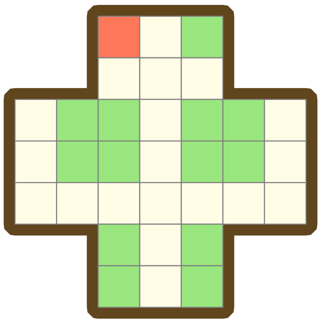


Figure 6.4: Visualization of solution vector for TASK1.

this level of symmetry is unexpected. In fact, by adjusting the constraints so that the arbitrarily chosen potential values for initial and goal state are 1 and 0 instead of 0 and 1, the only occupied-atom is replaced by the free-atom in the same location, leading to perfect symmetry along the vertical axis. The peg solitaire literature provides a clear explanation for this curiosity.

6.2 Invariant Parity Counts

This exact argument for proving the unsolvability of peg solitaire problems has been described by Beasley (1985), but similar approaches were known much earlier. Berlekamp et al. (2004) present the same argument, but in different terms, and trace its origin back to a paper by Reiss (1857). Beasley (1985) uses parity arguments to divide peg solitaire states into 16 classes such that there is no sequence of legal moves to bring a position of one class to a position of another. Stating that an initial and goal position are not of the same class therefore proves unsolvability. These classes can be represented by what Beasley (1985) calls "invariant parity counts", of which the solution vector for TASK1 is one.

An invariant parity count is a set of marked locations on the board. The locations must be chosen so that no legal move can change the parity of the number of pegs in marked locations for any position. There are a total of 16 sets of locations that fulfill this condition, including the uninformative empty set. Beasley (1985) provides the following instructions to easily construct the patterns by hand:

1. Choose any 2×2 square on the board and construct the 16 permutations of marked and/or unmarked cells.
2. Extend the pattern along rows and columns according to the rules:
    a) If two adjacent cells are both marked or both unmarked, their two neighbors are unmarked.
    b) Else, if one is marked and one unmarked, their two neighbors are marked.

The 15 informative counts contain 9 variations on the set in Figure 6.4, which Beasley (1985) names "square" counts, and 6 variations on the "diagonal" counts, with an example shown in Figure 6.5. When evaluating a state, we count the number of pegs present in the marked

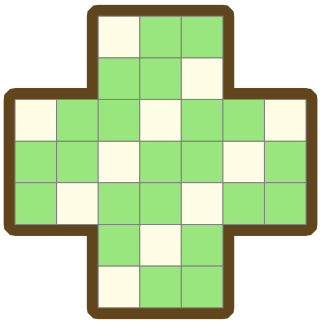


Figure 6.5: Example for invariant parity count of type "diagonal".

locations and note the number's parity. Two states are of the same class if and only if their parities are equal for all 16 invariant parity counts.

The argument our solution vector states is that the initial and goal state of TASK1 are of different parity with respect to the invariant parity count represented by the solution. They can therefore not be of the same class and the task is unsolvable.

6.3 Other Tasks

Our algorithm proves unsolvability for all except two pegsol instances in this way, with the only difference being that different variations of the square counts are used. Figure 6.6 shows the three other variations that were found. One of them is the solution for TASK1 with a

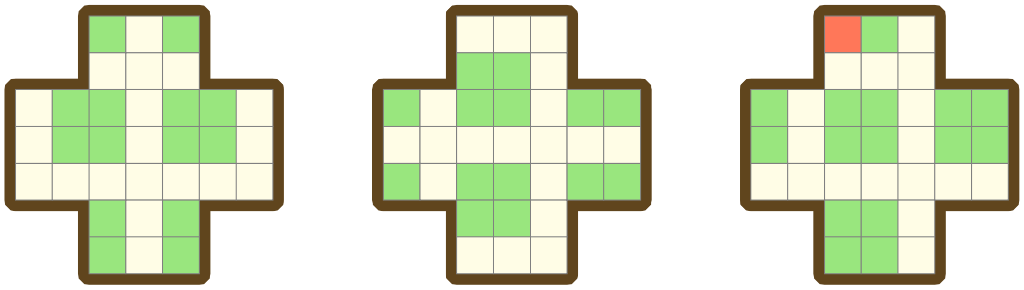


Figure 6.6: Three more invariant parity counts of type “square” found by our algorithm.

flipped occupied-atom, the other is a rotated version of the solution for TASK1, and the last one is a unique square count.

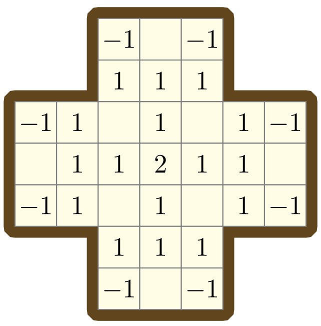
As mentioned, our algorithm fails to prove unsolvability for two identical[5] instances. This is due to the fact that initial and goal state belonging to different classes is a sufficient, but not necessary, condition for unsolvability. An example for an unsolvable task with equal class membership is the central game from Figure 6.2, where initial and goal state are reversed. This example shows that two states within a class cannot necessarily reach each other using legal moves, and gives us a hint towards the necessary change to make the statement true.

A state of class c can reach any other state in c using regular moves and *reverse* moves, where a peg jumps over an empty hole that is then filled with a peg. According to Beasley (1985), it is not possible to get more informative classifications of peg solitaire states using only parity arguments, and one must consider that moves consume pegs in order to gain further insights.

6.4 Numerical Unsolvability

Beasley (1985) considers consumed pegs through what he calls “resource counting”. He encodes the resources available on the board in a *resource count* which assigns a value to every cell. The values must be chosen such that for any horizontally or vertically adjacent group of three cells A, B and C, it must hold that $A + B \geq C$. An example from Beasley (1985) for a resource count is shown in Figure 6.7. Empty cells have a value of 0. The constraint

[5] Identical because our benchmark contains duplicate tasks in the pegsol domain.



		−1		−1		
		1	1	1		
−1	1		1		1	−1
	1	1	2	1	1	
−1	1		1		1	−1
		1	1	1		
		−1		−1		

Figure 6.7: Example for resource count.

ensures that no move can increase the resource amount. A state is evaluated by summing up the resource count values of all occupied holes, which allows us to formulate unsolvability arguments by stating that a problem's initial state contains fewer resources than its goal state. According to Beasley (1985), this argument was first made by John H. Conway and John M. Boardman in 1961, but no reference is given. We have previously come across Conway's argument when discussing the pegsol-row5 domain in Section 5.3, and learned that Conway and his co-authors call resource counts pagoda functions (Berlekamp et al., 2004). Finding such functions automatically has been attempted before, for example by Kiyomi and Matsui (2001), who formulate the problem as an integer program, which they relax and solve to prove unsolvability for peg solitaire problems.

Of course, the above argument is familiar to us as the generalization of our parity approach, with the difference that only one-dimensional features are considered. Resource counts (and equally pagoda functions) can thus be seen as monotonic separating functions limited to one-dimensional features.

Interestingly, resource counts are also not sufficient to prove unsolvability for the specific task for which our algorithm failed, which is shown in Figure 6.8.

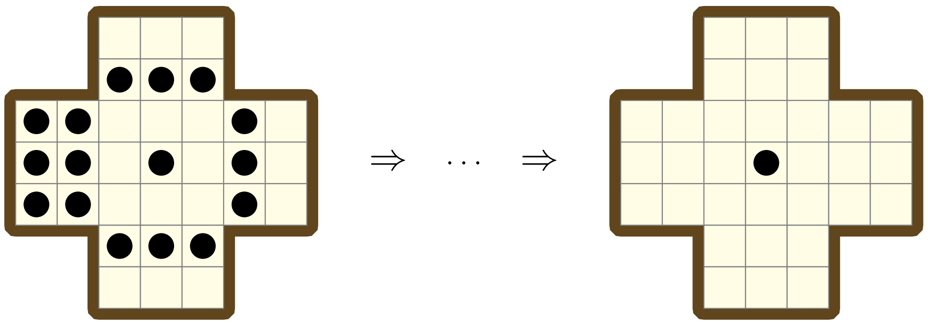


Figure 6.8: Task which our algorithm and resource count fail to prove unsolvable.

Proposition 6. *There exists no resource count R for planning task Π, pictured in Figure 6.8, such that $R(s_0) < R(s_*)$.*

Proof. For adjacent values A, B and C of R we can show that B cannot be negative because both $A+B \geq C$ and $C+B \geq A$ must hold (Beasley, 1985). This implies that only cells that cannot be jumped over can have negative values, and that $R(s_*) \geq 0$. Because the middle peg is the only peg in s_* and is also present in s_0, it holds that $R(s_0) = R(s_*) + R(s_{rest})$, where s_{rest} is the initial state without the middle peg. For $R(s_0) < R(s_*)$ to be true, $R(s_{rest})$ must be negative, with the only possible negative components of $R(s_{rest})$ coming from the leftmost pegs of the rows just below and above the middle row. Let us now consider the values at these cells as a, with b and c as their neighbors to the right. Assigning a to $-x$ binds b to be $\geq x$, because c cannot be negative and $a + b \geq c$ must hold. Any negative contribution by the possible pegs a is therefore compensated by pegs in b, which are both present in s_0. We have thus shown that $R(s_{rest})$ can never be negative and the proposition must hold. □

Beasley (1985) states that resource counts would fully describe the solvability of a peg solitaire game where fractional moves, and any number of pegs per hole are allowed. He goes on to discuss more in-depth unsolvability detection techniques, that go beyond the scope of this thesis.

7 Conclusion

The aim of this thesis was to explore the automatic generation of parity arguments as a way to detect unsolvability in planning tasks. Existing techniques involving potential functions that encode monotonic separating functions have been described by Pommerening (2017), and applied to the unsolvability planner Aidos by Seipp et al. (2016). We limited this real-valued approach to the field F_2, which provides a natural way to express parity arguments.

We saw that parity arguments can be expressed in terms of an s_0-invariant I, with the additional condition that I does not hold in any goal state. Potential functions are capable of combining invariant and condition into a single function, so that such a parity function's existence is sufficient to prove a planning task unsolvable.

To compute parity functions automatically, we constructed a set of constraints which guarantees that any potential function satisfying them is a parity function. We showed that these constraints can be represented efficiently for up to two-dimensional features, and how they can be constructed in practice.

In order to test the practical use of the resulting algorithm, we implemented it on top of the Fast Downward planning system. Working in F_2 proved beneficial to this stage, where it allowed us to replace complex LP solvers with more predictable Gaussian elimination. We compared the performance of three implementations on a benchmark containing unsolvable tasks from 19 domains. While we were able to improve performance with respect to resource consumption, it did not lead to more unsolvability proofs. Our approach seems to only be useful for a limited number of domains, with pegsol and sliding-tiles being the only two domains where we found parity arguments. Because our algorithm performed exceptionally well in the sliding-tiles domain, we ran experiments with bigger tasks and saw that even the very large 24 puzzle can be proven unsolvable within reasonable resource bounds.

In the last part of the evaluation, we compared our approach to the aforementioned unsolvability planner Aidos, as well as to the dead-end potentials component of its portfolio. While both configurations vastly outperform our algorithm over all, parity arguments seem very strong for sliding-tiles and are able to find unsolvability proofs in pegsol that are not detected by dead-end potentials.

Lastly, we took a closer look at actual parity arguments our algorithm found for the pegsol domain. We were able to show that equivalent proofs have been described in literature, and investigated the expressive power of parity and monotonic separating functions for this specific application.

7.1 Future Work

In this section we briefly discuss possible angles for further research in connection with the topics discussed in this thesis.

7.1.1 Gaussian Elimination Alternatives

While Gaussian elimination is a natural choice for solving the XOR constraints posed by our approach, it is not the only one. It has a time complexity of $\mathcal{O}(n^3)$ for a matrix of size $n \times n$, which can be problematic considering how large the constraint matrices can grow.

A classic improvement upon Gaussian elimination is the result by Strassen (1968), which introduces an algorithm for matrix multiplication that can be applied to solving systems of linear equations, and has a time complexity of $\mathcal{O}(n^{log_2 7}) \approx \mathcal{O}(n^{2.807})$. Bard (2007) shows how this approach can be applied to F_2.

Similarly, the Method of Four Russians for Inversion proposed by Bard (2007) describes another alternative, based on an algorithm by Arlazarov et al. (1970). It can achieve the same result as Gaussian elimination while reducing time complexity to $\mathcal{O}(n^3/\log n)$.

A third option that may come to mind is SAT solvers. It is possible to convert XOR constraints to conjunctive normal form, which allows SAT solvers to interpret the problem. Unfortunately this approach is limited by the fact that the necessary conversion generally increases the constraint size exponentially. Nevertheless, much research effort has been invested into integrating XOR constraints with SAT solvers. Some examples are Warners and van Maaren (1998), Baumgartner and Massacci (2000), and Soos et al. (2009). This is often done in the context of cryptography where XOR constraints are common. The findings of Soos et al. (2009) have been implemented as an extension to the MiniSat solver by Eén and Sörensson (2004). The result is CryptoMiniSat[6]. It offers native support for XOR constraints and solves them using a combination of SAT-solving concepts and Gaussian elimination. Possible issues with this approach would be internal optimizations that can include heuristics or even random decisions, which make results less predictable. Even more importantly, we are unsure if such solutions offer a polynomial guarantee for solving XOR constraints.

7.1.2 Relation to Dead-end Potentials

We have seen pegsol tasks where our algorithm was able to find parity arguments and dead-end potentials were not successful. It would be interesting to analyze such cases in more detail, and to determine under what circumstances they arise. Such a study could provide valuable insights into the exact relation between parity arguments and their real-valued counterpart.

[6] https://github.com/msoos/cryptominisat

7.1.3 Theoretical Considerations

We have shown that any solution found by our approach is a two-dimensional separating function in F_2 over all states that do not violate h^2 mutexes. Given this statement, it is natural to wonder whether the converse also holds: if such a separating function exists, our approach can find it. While we have not discussed this, an answer would strengthen the fundamental theory behind our approach.

A similar question arises with the observation that our approach seems to be able to find a parity argument whenever it is given a h^2 mutex that proves the goal to be unreachable, without using this information directly. Understanding whether this is the case in general would lead to a clearer picture of the expressive power of our approach.

Parity arguments may not yet have reached their full potential in the context of unsolvability in planning. More general approaches are possible, for example allowing higher-dimensional features. Such a change could remove the need to include mutexes or lead to success in a broader range of domains.

We hope that this thesis can serve as a piece in the puzzle of unsolvability in planning, and we are looking forward to seeing the ever-growing picture develop.

Bibliography

Alcázar, V., and Torralba, Á. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 2–6. AAAI Press, 2015.

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. Revisiting regression in planning. In Rossi, F., editor, *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 2254–2260. AAAI Press, 2013.

Archer, A. F. A modern treatment of the 15 puzzle. *The American Mathematical Monthly*, 106(9):793–799, 1999.

Arlazarov, V. L.; Dinitz, Y. A.; Kronrod, M. A.; and Faradzhev, I. A. On economical construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194(3):487–488, 1970. In Russian, English Translation in Soviet Mathematics Doklady.

Bäckström, C., and Nebel, B. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Bäckström, C.; Jonsson, P.; and Ståhlberg, S. Fast detection of unsolvable planning instances using local consistency. In Helmert, M., and Röger, G., editors, *Proceedings of the Sixth International Symposium on Combinatorial Search (SoCS 2013)*, pages 29–37. AAAI Press, 2013.

Bard, G. V. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Application to Cryptanalysis.* PhD thesis, University of Maryland, 2007.

Baumgartner, P., and Massacci, F. The taming of the (X)OR. In Llyod, J.; Dahl, V.; Furbach, U.; Kerber, M.; Lau, K.-K.; Palamidessi, C.; Sagiv, Y.; and Stuckey, P. J., editors, *Computational Logic - CL 2000*, volume 1861 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2000.

Beasley, J. D. *The Ins & Outs of Peg Solitaire (Recreations in Mathematics).* Oxford University Press, 1985.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. *Winning Ways for Your Mathematical Plays*, volume 4. A K Peters, 2 edition, 2004.

Blum, A. L., and Furst, M. L. Fast planning through planning graph analysis. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995)*, pages 1636–1642. Morgan Kaufmann Publishers Inc., 1995.

Bonet, B., and Geffner, H. Planning as heuristic search. *Artificial Intelligence*, 129(1–2): 5–33, 2001.

Chen, Y.; Xing, Z.; and Zhang, W. Long-distance mutual exclusion for propositional planning. In Sangal, R.; Mehta, H.; and Bagga, R. K., editors, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1840–1845. Morgan Kaufmann Publishers Inc., 2007.

Chung, F.; Graham, R.; Morrison, J.; and Odlyzko, A. Pebbling a chessboard. *The American Mathematical Monthly*, 102(2):113–123, 1995.

Eén, N., and Sörensson, N. An extensible SAT-solver. In Giunchiglia, E., and Tacchella, A., editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

Eriksson, S. Unsolvable PDDL benchmarks. https://doi.org/10.5281/zenodo.3355446, 2019.

Eriksson, S.; Röger, G.; and Helmert, M. Unsolvability certificates for classical planning. In Smith, S. F.; Barbulescu, L.; Frank, J.; and Mausam, editors, *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pages 88–97. AAAI Press, 2017.

Fišer, D.; Horčík, R.; and Komenda, A. Strengthening potential heuristics with mutexes and disambiguations. In Beck, J. C.; Buffet, O.; Hoffmann, J.; Karpas, E.; and Sohrabi, S., editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pages 124–133. AAAI Press, 2020.

Fox, M., and Long, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Gerevini, A., and Schubert, L. Inferring state constraints for domain-independent planning. In Mostow, J., and Rich, C., editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAi 1998)*, pages 905–912. AAAI Press, 1998.

Haslum, P., and Geffner, H. Admissible heuristics for optimal planning. In Chien, S. A.; Kambhampati, S.; and Knoblock, C. A., editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, pages 607–613. AAAI Press, 2000.

Helmert, M. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Hoffmann, J.; Kissmann, P.; and Torralba, Á. "Distance"? Who cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., editors, *Proceedings of the Twenty-First European Conference on Artificial Intelligence (ECAI 2014)*, pages 441–446. IOS Press, 2014.

Johnson, W. W., and Story, W. E. Notes on the 15 puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

Kiyomi, M., and Matsui, T. Integer programming based algorithms for peg solitaire problems. In Marsland, T., and Frank, I., editors, *Computers and Games - CG 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2001.

Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Korf, R. E., and Felner, A. Disjoint pattern database heuristics. *Artificial Intelligence*, 134 (1–2):9–22, 2002.

Lipovetzky, N.; Muise, C.; and Geffner, H. Traps, invariants, and dead-ends. In Coles, A.; Magazzeni, D.; Coles, A.; Edelkamp, S.; and Sanner, S., editors, *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2016)*, pages 211–215. AAAI Press, 2016.

Pommerening, F. *New Perspectives on Cost Partitioning for Optimal Classical Planning.* PhD thesis, University of Basel, 2017.

Pommerening, F., and Helmert, M. A normal form for classical planning tasks. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 188–192. AAAI Press, 2015.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. From non-negative to general operator cost partitioning. In Zilberstein, S.; Bonet, B.; and Koenig, S., editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 3335–3341. AAAI Press, 2015.

Pommerening, F.; Helmert, M.; and Bonet, B. Higher-dimensional potential heuristics for optimal classical planning. In Zilberstein, S.; Singh, S.; and Markovitch, S., editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*, pages 3636–3634. AAAI Press, 2017.

Reiss, M. Beiträge zur Theorie des Solitär-Spiels. *Crelles Journal*, 54:344–379, 1857.

Rintanen, J. An iterative algorithm for synthesizing invariants. In Kautz, H., and Porter, B., editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)*, pages 806–811. AAAI Press, 2000.

Seipp, J.; Pommerening, F.; and Helmert, M. New optimization functions for potential heuristics. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 193–201. AAAI Press, 2015.

Seipp, J.; Pommerening, F.; Sievers, S.; Wehrle, M.; Fawcett, C.; and Alkhazraji, Y. Fast Downward Aidos. *Unsolvability International Planning Competition: planner abstracts*, pages 28–38, 2016.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

Soos, M.; Nohl, K.; and Castelluccia, C. Extending SAT solvers to cryptographic problems. In Kullmann, O., editor, *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

Strassen, V. Gaussion elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1968.

Warners, J. P., and van Maaren, H. A two-phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23(3-5):81–88, 1998.